

Abstraction and Abstraction Refinement in the Verification of Graph Transformation Systems

Vom Fachbereich Ingenieurwissenschaften
Abteilung Informatik und angewandte Kognitionswissenschaft
der Universität Duisburg-Essen
zur Erlangung des akademischen Grades eines
Doktor der Naturwissenschaften (Dr.-rer. nat.)
genehmigte Dissertation

von
Vitaly Kozyura
aus Magadan, Russland

Referent: Prof. Dr. Barbara König

Korreferent: Prof. Dr. Arend Rensink

Tag der mündlichen Prüfung: 05.08.2009

Abstract

Graph transformation systems (GTSs) form a natural and convenient specification language which is used for modelling concurrent and distributed systems with dynamic topologies. These can be, for example, network and Internet protocols, mobile processes with dynamic behavior and dynamic pointer structures in programming languages. All this, together with the possibility to visualize and explain system behavior using graphical methods, makes GTSs a well-suited formalism for the specification of complex dynamic distributed systems.

Under these circumstances the problem of checking whether a certain property of GTSs holds – the verification problem – is considered to be a very important question. Unfortunately the verification of GTSs is in general undecidable because of the Turing-completeness of GTSs. In the last few years a technique for analysing GTSs based on approximation by Petri graphs has been developed. Petri graphs are Petri nets having additional graph structure.

In this work we focus on the verification techniques based on counterexample-guided abstraction refinement (CEGAR approach). It starts with a coarse initial over-approximation of a system and an obtained counterexample. If the counterexample is spurious then one starts a refinement procedure of the approximation, based on the structure of the counterexample. The CEGAR approach has proved to be very successful for the verification of systems based on their over-approximations.

This thesis investigates a counterexample-guided abstraction refinement approach for systems modelled with GTSs. Starting with a given spurious counterexample, we describe here how to construct a more exact approximation (by separating merged nodes) for which this counterexamples disappears. This procedure can be performed repeatedly for any number of spurious counterexamples. Furthermore, an incremental coverability approach for Petri nets is developed, which allows one to speed-up the construction of over-approximations of GTSs.

A well-known approach is to extend a modelling language with the possibility of describing attributes as values of some data types. The approximation-based verification technique, including a counterexample-guided abstraction refinement, is hence also generalized in this work to attributed GTSs (AGTSs), where the attributes are abstracted in the framework of abstract interpretation.

In the practical part, a verification tool AUGUR 2 is developed, which supports the whole verification process for GTSs and AGTSs. A number of case studies (both attributed and non-attributed GTSs) were successfully solved with AUGUR 2.

Acknowledgement

First of all, I would like to thank my doctoral adviser, Barbara König, for being my supervisor, for encouraging and challenging me throughout the academic program. This work would not have been possible without her. I want to express my gratitude to Javier Esparza, whose works on Petri net unfolding have awakened my interests to the related areas of computer science. I thank him also for his support and advices on various aspects of my work.

I also would like to thank my colleagues, Sander Bruggink, Tobias Heindel, Stefan Kiefer, Michael Luttenberger, Claus Schröter, Alin Stefanescu and Dejavuth Suwimon-teerabuth, for their help and many interesting discussions on the topics of this thesis. Also I thank a lot all students have being involved in the development of the verification tool AUGUR. It was a very interesting and pleasant team work experience.

I am very grateful to my family, to my parents for their lifelong support, and to my wife for being my friend and companion.

Finally my thanks go to the reviewers of this thesis for reading the result of my efforts.

Contents

1	Introduction	3
1.1	Verification of Graph Transformation Systems	3
1.2	Summary and Contribution of the Thesis	5
1.3	Related Work	7
1.4	Publications	10
1.5	Contributions	12
2	Petri Nets	13
2.1	Introduction to Petri Nets	13
2.2	Coverability Graphs	14
2.3	Incremental Calculation of Coverability Graphs	18
2.4	Backward Coverability Algorithm	23
3	Graph Transformation Systems	26
3.1	Introduction to Graph Transformation Systems	26
3.2	Verification of Graph Transformation Systems	28
4	Attributed Petri Nets and Graph Transformation Systems	37
4.1	Algebras and Abstraction	37
4.2	Attributed Petri Nets	40
4.3	Attributed Graph Transformation Systems	42
4.4	Verification of Attributed Graph Transformation Systems	45
5	Counterexample-Guided Abstraction Refinement	53
5.1	Abstraction Refinement of Graph Transformation Systems	53
5.1.1	Spurious Runs	54
5.1.2	Relations on Nodes for Refining Abstract Runs	55
5.1.3	Elimination of Spurious Runs	58
5.1.4	Correctness	60
5.2	Abstraction Refinement of Attributed Graph Transformation Systems . .	63
6	Augur – a Tool for the Analysis of Graph Transformation Systems	69
6.1	Brief Description of AUGUR 1	69
6.2	Software Design of AUGUR 2	70
6.3	System Architecture of AUGUR 2	72
6.4	Functionality and Usage	77

7	Case Studies	83
7.1	Public-Private Servers	84
7.2	Firewall	87
7.3	Experiments with the Incremental Coverability Approach	91
7.4	Random Graph Transformation Systems - Statistical Results	93
7.5	Verifying Red-Black Trees	96
7.6	Leader Election Protocol	102
7.7	Needham-Schroeder Protocol	104
8	Conclusion and Future Work	108
A	Basic Category Theory for Graph Rewriting	118
B	Example in GTXL format	122
C	Example in new GTXL format	124
D	Example in GXL format	127
E	Example of an SPL program	129
F	Verification Example	131

Chapter 1

Introduction

1.1 Verification of Graph Transformation Systems

In the last decades, the development of computers and networks, the appearance and the rapid expansion of Internet technologies have led to increasing complexity and strong interconnection of computer systems. Under these circumstances, the problem of modelling and analyzing complex distributed systems is considered to be a very important question. Unfortunately, many of today's methods describe distributed and mobile systems with formalisms that either have an infinite state space, making many interesting questions undecidable, or a very a large state space, leading to the so-called state explosion problem. This makes the important problem of modelling and verification of mobile and distributed systems a very difficult one.

The approach in this thesis is based on a rather natural and expressive modelling language, namely graph transformation systems (GTSs) [95]. GTSs have their origin in the late 1960's and were developed up to now into a rich theory with many different branches. The development was originally motivated by such areas of computer science as pattern recognition, description of data types and compiler construction. Later, more modern branches of computer science, such as computer networks with different topologies, mobile processes, UML diagrams and web services have also been specified and analysed with GTSs. Being a natural generalization of formal language theory, based on string rewriting and the theory of term rewriting, GTSs are also interesting from a theoretical point of view.

For many structures in computer science, a static description of system states can be obtained with the help of graphs in a rather natural way. As an example, we consider computer networks, where stations can be represented as nodes of graphs and the network connections as edges. As a more specific example, we mention here pointer structures on the heap in a programming language, where objects are nodes and the values of pointers are modelled with edges. These can be, for example, memory allocation structures in the programming language C, but the same structures arise also for object-oriented languages, such as Java.

Graph transformations provide also the possibility of describing the dynamical behavior of systems. Using GTSs one can model in a natural way dynamic creation and deletion of objects, for example, computer stations entering and leaving the network or mobile processes moving between different stations. GTSs are also a very convenient modelling language for systems with evolving topologies, where not only the number of the components but also the structure of the connections between them is dynamic. This is often the case in network and Internet protocols. Also in the evaluation of dynamic pointer structures on the heap in a programming language mentioned above, the topology

of the structures is a subject of constant changes.

The language of GTSs is also very helpful in the modelling of distributed and concurrent systems. Such systems appear as a natural result of using networks, and the research on their specification and verification is a large and important area in modern computer science. Distributed and concurrent systems are those systems where the execution of tasks and programs is divided between several processors or distributed across a network. Concurrent systems are related to the area of parallel computing, but in the case of concurrent systems the focus is more on the interactions between processes. GTSs are very suitable for the modelling of such systems, because of their ability to change different parts of graphs independently.

Another very useful feature of GTSs is that states and the behavior of complex systems can be visualized and explained using graphical methods, which makes the human perception of such systems much easier. All this makes GTSs a well-suited and expressive formalism for the specification of complex distributed dynamic systems on an intuitive level.

Verification of GTSs remains undecidable in general because of the Turing-completeness of GTSs (GTSs can easily simulate Turing machines) and in the last few years a technique for analysing GTSs based on approximations has been developed [12, 6]. GTSs are approximated by Petri graphs, which are Petri nets [85] having additional graph structure. Petri graphs can be seen as symbolic representations of graph transition systems and also as representations of sets of abstract runs. Petri nets can then be analyzed with standard verification techniques [85].

The Petri net formalism is widely used for modelling concurrent systems. Petri nets have first appeared in the works of C.A. Petri in 1962 and since that time they were developed to one of the most important formalisms for the modelling and verification of distributed and concurrent systems. This has happened due to the decidability of many interesting questions for Petri nets, combined with the expressiveness and natural presentation of many classes of distributed systems.

This thesis is not focused on the description and analysis of Petri nets themselves, although some of the developed methods can be useful for the analysis of Petri nets in general form. We consider Petri nets only in the context of GTSs as a suitable and expressible approximation formalism. In [12, 6] it was also shown that it is often possible to verify properties of GTSs (which are in general undecidable) based on over-approximating Petri graphs. If the property of GTSs is verified by means of Petri graphs, then we have successfully solved the problem. On the other hand, refinement of the over-approximation can sometimes be needed.

Verification techniques based on counterexample-guided abstraction refinement (CEGAR approach) [24] have proved to be very successful for the verification of systems and their approximations. The idea behind this approach is to start with a coarse initial abstraction or over-approximation of a system and to check whether a certain property can be verified using this abstraction. If it cannot be verified, one obtains a run in the approximation that violates the property, also called counterexample. Now either this counterexample is real or it is spurious, i.e., it has been introduced by the approximation. In the latter case the approximation is refined in such a way that the counterexample disappears. This process is repeated. In the case of infinite-state systems, however, there is in general no guarantee that it will terminate, since the properties to be verified are usually undecidable.

- ✓ The first theoretical challenge of this thesis is to investigate a counterexample-guided abstraction refinement approach in the framework of GTSs.

A well-known and widely used approach is to extend a modelling language by adding data types and suitable operations (usually as attributes). This is for instance done in

coloured Petri nets [57], algebraic high level nets [39], and attributed GTs (AGTs) [61, 104, 78]. Extending GTs with attributes allows one to combine the intuitive graphical aspects of the modelled systems with natural data structures, which makes such extended GTs more suitable for practical applications. In addition this leads to more compact models (because for many well-known operations it is no longer necessary to describe them in some artificial way by graph structures).

- ✓ The second theoretical purpose of this thesis is the development of an approximation-based verification technique for AGTs (including also a counterexample-guided abstraction refinement approach).

Because the data types are often infinite, abstraction of data types is needed. This is usually done in one of the following ways: data abstraction [25], abstract interpretation [103, 28] or predicate abstraction [47, 31, 50, 59]. In this work we concentrate us on the first two approaches. The usual way in abstract interpretation to describe the abstraction of data structures and the operations on them is to use the theory of Galois connections [2]. The predicate abstraction approach for AGTs remains a topic for future work.

- ✓ The practical challenge of this thesis is the development of a software tool (called AUGUR 2) supporting the whole process of the verification of both GTs and AGTs.

The tool allows one to solve a number of case studies, modelled by both attributed and non-attributed GTs.

1.2 Summary and Contribution of the Thesis

Below we discuss the structure and the results of the thesis.

Chapter 2 (Introduction to Petri Nets). In this chapter we first give definitions and a brief description of Petri nets (Section 2.1). Then in Section 2.2 we describe the standard technique for checking the coverability property for Petri nets by constructing a coverability graph. Coverability checking plays an important role in the construction and analysis of over-approximations of GTs (Section 3.2).

The next section (Section 2.3) is the first result of this thesis, an incremental coverability approach for Petri nets. Furthermore, in Section 7.3 it will be shown how to speed-up the construction of over-approximations of GTs using the incremental approach to the coverability problem. The construction of the coverability graph can be seen as an approach with a forward search for the covered marking. The last section (Section 2.4) describes another (backward) approach to the coverability problem.

Chapter 3 (Graph Transformation Systems). This chapter gives a brief introduction to GTs (Section 3.1) and describes the technique for their static verification based on over-approximating GTs by Petri graphs. Although in the description of GTs we focus on the categorical DPO (double-pushout) approach to graph rewriting [95], the chapter presents a rather low-level non-categorical description of GTs and its over-approximations. We justify this by the practical aspect of the thesis, namely, the software implementation of the verification techniques (Chapter 6).

Chapter 4 (Attributed Petri Nets and Graph Transformation Systems). In Section 4.1 we introduce the algebra of attributes and the attribute abstraction. Then we describe our view on attributed Petri nets and attributed GTs (AGTs) (Sections 4.2 and 4.3), which is based on the algebraic specification approach to the specification of attributes as data types. We develop our description according to our demands on the analysis and verification procedures for AGTs. The developed technique for

over-approximation and verification of AGTSs is presented in Section 4.4. We describe the abstraction of attributes in the manner of abstract interpretation and connect this description with our algebraic specification of AGTSs.

Having approximated AGTSs by attributed Petri graphs and also having abstracted the attributes to finite domains, we can reduce the problem of verification of AGTSs to the analysis of the obtained attributed Petri nets. Some of these analysis techniques are described in Section 4.2, where we adapt the coverability techniques for Petri nets from Chapter 2 to attributed Petri nets, which allows us to verify approximations of AGTSs.

Chapter 5 (Counterexample-Guided Abstraction Refinement). This chapter presents the main theoretical result of the thesis, namely the theory of counterexample-guided abstraction refinement for GTs (Section 5.1) and for AGTSs (Section 5.2).

We consider in Section 5.1 the notion of spurious run for GTs and show how to find the reason for the appearance of the spurious run (spurious counterexample), which is usually the merging of some nodes. Then we show how one can construct a new refined unfolding where the given counterexample is eliminated, and we prove the necessary theoretical results.

The theory in Section 5.2 for AGTSs is more complex, because in this case we often also have the necessity to refine the abstraction of attributes. In the theory of structural refinement (refinement of a hypergraph structure) we follow Section 5.1, adapting the developed theory to the case of AGTSs.

Chapter 6 (Augur – a Tool for the Analysis of Graph Transformation Systems). In the scope of this work an important achievement is the design and implementation of a software tool called AUGUR 2. Section 6.1 describes briefly the previous version of the tool (AUGUR 1) and the problems and the demands which have led to the necessity of a completely new version of the tool.

In Section 6.2 we discuss the background concepts of the software design of the new tool, which allows us to obtain a flexible and extendable software environment. The main components and modules of the tool are discussed in Section 6.3. In the last section (Section 6.4) we shortly describe the functionality and usage of AUGUR 2 as a command line tool and also the graphical user interface (GUI).

Chapter 7 (Case Studies). In this chapter we discuss some case studies, which were successfully solved using AUGUR 2. The first two examples, namely the Public-Private Server (Section 7.1) and the Firewall Example (Section 7.2) are non-attributed systems, where the over-approximations need to be refined in order to obtain the verification result. We also compare the counterexample-guided abstraction refinement approach to another refinement possibility, namely the depth-based approach based on the construction of an over-approximation exact up to a certain depth (Section 3.1).

In the next section (Section 7.3) we describe our approach and the experimental results concerning the optimization of the unfolding procedure based on the incremental coverability technique for Petri nets (Section 2.3). Section 7.4 is dedicated to some statistical results obtained for our verification techniques. In order to gather the statistical material we generate random graph transformation systems.

In Section 7.5 we show how to verify the correctness of insertion of elements into red-black trees, a form of balanced search trees, using analysis techniques developed for GTs. We first model red-black trees and operations on them using hypergraph rewriting. Then we use AUGUR 2 in order to show that insertion preserves the property that there are no two consecutive red nodes in a tree, a requirement for red-black trees.

The last two case studies describe the verification of AGTSs. In Section 7.6 we model and verify a leader election protocol in the ring topology and in Section 7.7 we model the well-known attack on the Needham-Schroeder protocol in the framework of AGTSs.

1.3 Related Work

Analysis of graph transformation systems is a rather recent area of research and there exists relatively little work on their verification and analysis so far. This fact can be partially explained by the complexity of analysis of GTSs and therefore, most research in this area was made either in the framework of semantic issues (rewriting formalisms, concurrency, expressiveness) or in the practical applications such as software engineering.

Early results in the analysis of GTSs and specific temporal logics can be found in [63] and [46]. In these papers temporal logic was enriched with graphical components in order to make it possible to specify the behaviour of GTSs. The obtained logic has been used for specification and verification of safety and liveness properties of GTSs. Also the compositional operational semantics for GTSs was developed, which integrates the single-pushout approach [95] to GTSs and a propositional temporal logic. In [48] a compositional method for verifying of GTSs was proposed. For analysis purposes the notion of *view* was introduced, which is an incomplete specification and describes only some aspects of the GTS. A view can be considered as an under-approximation of the behavior of the complete system. This means that properties of the view are inherited by the original GTS.

In more recent research one can divide between works concerned with an analysis of finite-state GTSs and works on the analysis of infinite-state GTSs. In the finite case some research groups [108, 32, 87, 92] follow the idea of using a model-checking technique [26] with an input obtained from the translation of GTSs. In [108] GTSs are considered as a visual specification paradigm describing the operational semantics of meta-modeling languages such as the Unified Modeling Language (UML)¹. GTSs are automatically transformed into SAL specifications [21] which are used as an input for symbolic analysis techniques. In [32] a visual specification language called Object-Based Graph Grammars (OBGG) is developed in order to specify asynchronous distributed systems. The obtained OBGG are then translated into the PROMELA language² and verified using model checking techniques. The paper [87] describes work on the generation of transition systems from GTSs and checking their properties expressed in a temporal logic on graphs. GTSs are considered as a behavioural semantics of object-oriented programs. In [92] two different approaches to the model checking of systems described by GTSs are compared: The first approach is based on encoding graphs into fixed state vectors and transformation rules into guarded commands. This allows quick integration into existing model checking tools. The second approach is to simulate the rules of GTSs directly and build the corresponding state space.

In the analysis of infinite-state GTSs we could mention the works [88, 89, 91, 19] on abstraction using shape graphs and other abstract representations of GTSs. In [88] a decidable fragment of first-order graph logic (local shape logic (LSL)) is proposed, which is used for abstracting GTSs by reasoning not about individual state graphs, and also the notion of canonical shape of a state graph is defined. The work [89] approximates edge-labelled graphs by considering only the local structure of the graphs, namely for each node of the graph only the approximate number of its neighbours is remembered. This allows one to obtain a finite approximation. In [91] graphs are approximated by collecting nodes that are sufficiently similar and the application of rules is described on the abstract level. This also results in smaller states and a finite state space. We mention here also the work [18] on the analysis of partner graph grammars. For partner graph grammars a two-layered abstraction is proposed and formally proved sound. In [19] the authors present a new approach to abstract graph transformation. The abstractions introduced in [88] and [18] are generalised in this work and put into a framework of

¹<http://www.uml.org/>

²<http://www.spinroot.com/spin/Man/promela.html>

neighbourhood abstraction. In the neighbourhood abstraction nodes are summarised if they have similar neighbourhood (up to some radius), which enables both abstraction and abstraction refinement (by the increasing of the radius). Also a modal logic for analysis of graphs based on their abstractions is introduced.

The basis for this thesis are the works [6, 12, 13] on analysis of infinite-state GTSs by approximating them with Petri graphs. In the paper [6] the following static analysis technique for GTSs is introduced: Given a GTS an approximating unfolding algorithm produces a finite structure, called Petri graph, consisting of a Petri net and a hypergraph structure over it. Petri graphs can be seen as an approximation of the Winskel style unfolding of the GTS. In the work [12] the notion of over-approximating Petri graph was extended to the more general notion of k -covering and it was shown how to check safety and liveness properties of a GTS on the obtained k -covering. A monadic second-order logic over graphs was introduced in [13], which is expressive enough to characterise typical graph properties. It was shown how the formulae can be effectively verified by the over-approximating Petri graph. Based on the approximating Petri graphs also a framework for finite-state systems modelled as GTSs was developed in [9]. Here a technique based on unfolding semantics, which generalises McMillans complete prefix approach [81, 42], originally developed for Petri nets, was transferred to GTSs. It was also shown that the properties of the graphs reachable in the system, expressed in a monadic second-order logic, can be verified using the obtained complete prefix.

Whereas in the case of finite-state GTSs the standard model-checking tools (such as SPIN [55] or SMV [62]) are often used, there are also special tools which were written for the analysis of GTSs. For example the tool GROOVE was developed (see [61]), where graphs are used for both design-time and run-time models of software systems. We mention here also the Graph Backwards Tool (GBT)³, where the symbolic backward reachability procedure on graphs is implemented. Besides the tools mentioned above the following tools are related to the specification and analysis of GTSs: PROGRES, a graph grammar programming environment [101, 102]; GenGED, a visual editor based on GTSs [15]; DiaGen, a system for producing of the diagram editors for visual languages [83].

A related area to the verification of GTSs is the theory of finding over-approximations of pointer structures, also known as shape analysis. One idea is to consider these over-approximations as being a model of a 3-valued logic [97]. The technique is not fully automatic and some steps concerned with predicates and predicate transformers have to be done manually. Compared to shape analysis, which is also concerned with over-approximation techniques for graphical structures and which represents these structures as models of a 3-valued logic, we follow a different approach where graphs are represented directly and graph morphisms are used as a convenient abstraction mechanism. Furthermore, approximations with Petri nets enable us to talk about multiplicities of edges, and they can be conveniently analyzed using a variety of existing Petri net tools.

This thesis deals with an analysis of infinite-state GTSs and our first task was to investigate a counterexample-guided abstraction refinement approach in the analysis of GTSs based on their approximations by Petri graphs. Counterexample-guided abstraction refinement has its origin in [24] and is now a subject of an intensive investigation. The technique has been used successfully in several tools such as SDV (Static Driver Verifier) [14], BLAST (Berkeley Lazy Abstraction Software Verification Tool) [50] or MAGIC (Modular Analysis of proGrams In C) [23].

The following work has been done concerning abstraction refinement for graph-like structures: In [76] models of a 3-valued logic representing pointer structures are refined in the framework of shape analysis by generating new instrumentation relations. In [12] a chain of finite over-approximations (k -coverings) of the unfolding is constructed

³<http://www.it.uu.se/research/group/mobility/adhoc/gbt/>

and it is proved that the chain converges to the complete unfolding. Finally, in [19] increasing the radius in the neighbourhood abstraction can be considered as a framework for abstraction refinement. We are not aware of any work on counterexample-based abstraction refinement for graph-like structures.

The next task of this thesis was to develop an analysis technique for GTSSs extended with attributes (AGTSSs). As an example of modelling languages extended with attribute specification we call here coloured Petri nets [57]. We mention here the following works concerned with a description and an analysis of AGTSSs: the AGG (the Attributed Graph Grammar System) tool [104], which has a visual environment for visualization and analysis of AGTSSs, the work with AGTSSs in the frame of the project GROOVE [61], the development of the algebraical framework for AGTSSs in [77, 78] and the theoretical categorical approach to AGTSSs [38].

Because we use an algebraic approach for the description of AGTSSs, we also mention here the works [39, 52, 37], where the language of Petri nets is combined with algebraic specifications leading to the concept of algebraic high-level nets (AHL-nets). Suitable compositionality results are shown and the obtained AHL-nets are represented in the framework of so-called AHL-net-transformation systems. This concept allows one to build AHL-nets from basic components and also to transform them using rules in the manner of GTSSs.

Concerning the verification of system models, we are usually interested only in specific system properties. In the standard model-checking approach, the properties are usually specified in some temporal logic such as linear time logic (LTL), branching time logic (CTL) or general modal logic such as modal μ -calculus [26]. The properties in AUGUR 2 can be described either in the form of regular expressions or as a logical formulas. The theory is based on the following works: [13] - introduction of a monadic second-order logic over graphs, [10] - a generalization of [13] by considering more expressive logics, where edge quantifiers and temporal modalities can be interleaved, [65] - analysis method checking for the absence of (Euler) paths or cycles in the set of reachable graphs by using the semilinear sets, [86] - analysis of the absence of forbidden paths by regular expressions.

Besides the undecidability of the verification problem for infinite-state GTSSs and AGTSSs in general, there are also the well-known state space explosion problem and the NP-hardness of some sub-procedures, which make the analysis of GTSSs and AGTSSs a very difficult task. This means that in this case it is important to do as much optimization as possible. One of the optimization directions is searching for better graph matching algorithms. The problem is NP-complete in the size of the matched graph, but still a number of heuristics have been proposed: [74, 96], using constraint satisfaction algorithms for efficient solving of the graph matching problem; [117], a search-plan-based graph matching technique used in the tool PROGRES; [109], an incremental approach to the graph matching problem; [17, 110], a heuristical matching strategy based on the cost model for possible matching strategies; [90], efficient solving of the isomorphism problem by using element-based graph certificate mappings in order to recognising non-isomorphic graphs. Some of this heuristics are implemented in AUGUR 2.

Other optimization possibilities are algorithms for analysing reachability and coverability properties of Petri nets. We mention here the following works: [85], standard coverability approach using the constructed coverability graph; [1], forward and backward coverability algorithms described in the general framework of well-structured transition systems (WSTSs) [45]; [35], application of an action planner to the verification of Petri nets. All this techniques have been used in the implementation of AUGUR 2.

AUGUR 2 works with different input and output formats: GTXL and GXL format [114, 53], a new version of the GTXL format [72], The AGG format [104] and a simple pointer language [105].

The following case studies have been successfully solved with AUGUR 1: the verifica-

tion of a mutual exclusion protocol [33] and the verification of red-black trees [5]. The following diploma and student works are parts of the tool AUGUR 1: the core unfolding algorithm [112], a description of properties by regular expressions [86], coverability algorithms [106], an efficient unfolding procedure [16], the generation of test cases described by GTSs [56].

In the framework of AUGUR 2 the following diploma and student works were written: translation of a simple pointer language into GTS [105], a description of properties by logical formulas and optimized coverability (reachability) analysis [111], efficient algorithms for the analysis of GTSs, especially for graph matching [116].

1.4 Publications

In this section we summarize the publications of the author and the author's contribution in them. The thesis is based on the following papers:

1. Barbara König and Vitali Kozioura. Counterexample-guided Abstraction Refinement for the Analysis of Graph Transformation Systems. In *Proc. of TACAS '06*, pages 197–211. Springer 2006. LNCS 3920.
2. Barbara König and Vitali Kozioura. Counterexample-guided Abstraction Refinement for the Analysis of Graph Transformation Systems. *Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik*, Technical Report 01. 2006.
3. Barbara König and Vitali Kozioura. Incremental Construction of Coverability Graphs. In *Information Processing Letters*, Volume 103(5) pages 203–209, 2007.
4. Vitali Kozioura. Verification of Random Graph Transformation Systems. In *Proc. of GT-VC 2006 (Workshop on Graph Transformation for Concurrency and Verification)*, Volume 175(4), pages 63–72, 2007. ENTCS.
5. Barbara König and Vitali Kozioura. Towards the Verification of Attributed Graph Transformation Systems. In *Proc. of ICGT '08 (International Conference on Graph Transformation)*. Springer, 2008. LNCS, to appear.
6. Barbara König and Vitali Kozioura. Augur—A Tool for the Analysis of Graph Transformation Systems. *EATCS Bulletin*, volume 87, pages 125–137, November 2005. Appeared in The Formal Specification Column.
7. Barbara König and Vitali Kozioura. Augur 2—A New Version of a Tool for the Analysis of Graph Transformation Systems. In *Proc. of GT-VMT '06 (Workshop on Graph Transformation and Visual Modeling Techniques)*, Volume 211, pages 201–210, ENTCS 2006.
8. Paolo Baldan and Andrea Corradini and Javier Esparza and Tobias Heindel and Barbara König and Vitali Kozioura. Verifying Red-Black Trees. In *Proc. of COS-MICAH '05*. 2005. Proceedings available as report RR-05-04 (Queen Mary, University of London).
9. Barbara König and Vitaly Kozyura. Case study: Verification of a leader election protocol using Augur. In *Solution for the GraBaTs '09 tool contest*. 2009.

In Papers 1 and 2 the author in cooperation with Barbara König proposed a counterexample-based abstraction refinement technique for (non-attributed) GTSSs. After describing the refinement algorithm, the correctness of the presented approach is proved and experimental results comparing it with earlier refinement technique for GTSSs (depth-based approach) are provided. The experiments have been done using the software tool AUGUR 1.

In Paper 3 the author and Barbara König have developed an incremental approach to the coverability problem of Petri nets. This approach is motivated by the over-approximation of GTSSs by Petri nets for verification purposes, where coverability checking in the intermediate steps leads to smaller and more exact approximations. The technique can also be useful for other applications where Petri nets are updated either interactively or automatically. The experimental results show that the usage of an incremental approach leads in many cases to faster construction of coverability graphs and the application of this theory to the construction of the over-approximating Petri graphs is in fact feasible.

Paper 4 describes the experiments of the author, which have been made in order to gather statistical results of the verification of GTSSs. For verification purposes the over-approximation approach extended with counterexample-guided abstraction refinement was used. The question here is how many GTSSs can be verified using the over-approximation technique and standard analyzing algorithms for Petri nets. The statistical results are quite positive and hence the verification of GTSSs by Petri graphs can be seen as a promising verification approach.

In Paper 5 written by the author in cooperation with Barbara König the approximation of attributed graph transformation systems (AGTSSs) by attributed Petri graphs is presented. A view on AGTSSs and attributed Petri graphs as labeled over a Σ -algebra is considered and the necessary correctness results are proved. Furthermore the counterexample-guided abstraction refinement approach developed earlier is extended to AGTSSs. The example of a leader election protocol shows the application of the developed theory to the verification of distributed systems.

In Paper 6 the first version of the tool AUGUR for verification of the GTSSs is presented. The author has participated in the development of this version by implementing the abstraction refinement module and by gathering all tool components in a common environment.

In the next version of the tool (AUGUR 2) presented in Paper 7 the author has developed a software design and proposed a software architecture of the tool. The purpose of the software design was to create a completely new tool having an open, flexible and easily extendable architecture. The author has supervised the implementation of the tool, which was partially done by students. The author himself has implemented the modules concerned with the theoretical part of the thesis: the abstraction refinement module, support for attributed GTSSs and its verification, and the incremental coverability algorithm.

In Paper 8 the author has participated in the experiments with red-black trees modeled by GTSSs and analyzed with the help of AUGUR 1.

Finally, in Paper 9 a case study of a leader election protocol is verified using the tool Augur. First a finite-state variant of the leader election protocol is investigated. It is shown how to verify it using McMillan style unfolding, which helps one to avoid an exponential state space explosion. Then a parametric version is considered, which is described with attributed graph transformation systems. This variant is verified using attributed Petri graphs as over-approximations in combination with a counterexample-guided abstraction refinement technique.

1.5 Contributions

In this section we summarize the contribution of the author of the thesis. We divide it into the research, implementation and experimental parts.

Research Results

The theoretical work is based on the over-approximation technique for GTSSs developed in [7]. The following research results were obtained by the author in cooperation with Barbara König.

1. A counterexample-based abstraction refinement approach for GTSSs (Section 5).
2. An incremental approach to the coverability problem of Petri nets, which allows faster construction of the over-approximating Petri graphs (Section 2.3).
3. An over-approximation based verification technique and a counterexample-based abstraction refinement approach for attributed GTSSs (Sections 4 and 5).

Implementation Results

Here we describe the implementation results achieved by the author.

1. Abstraction refinement module in AUGUR 1 (Section 6.1).
2. Software design and architecture of the tool AUGUR 2 (Sections 6.2 and 6.3).
3. Supervising the implementation of AUGUR 2.
4. Implementation of the following modules in AUGUR 2: abstraction refinement module, support for attributed GTSSs and its verification, incremental coverability technique (Section 6.3).

Experimental Results

The following experiments have been done by the author in the framework of the thesis:

1. Experiments with the firewall and public-private server examples in order to compare the counterexample-based and the depth-based abstraction refinement approaches (Sections 7.1 and 7.2)
2. Experiments proving the feasibility of an incremental approach to the calculation of a coverability graph (Section 7.3).
3. Experiments on the verification of random GTSSs by using the over-approximation technique (Section 7.4).
4. Participation in experiments on the verification of red-black trees modeled by GTSSs (Section 7.5).
5. Experiments on the verification of the leader election protocol and the Needham-Schroeder protocol modeled by attributed GTSSs (Sections 7.6 and 7.7).

Chapter 2

Petri Nets

In this chapter we briefly introduce the basic ideas behind the theory of Petri nets. The theory of Petri nets is not the main subject of this thesis and will be introduced only as a formalism used for the approximation of graph transformation systems (GTSs) (Chapter 3). A more detailed description of Petri net theory can be found, for example, in [85].

After a brief introduction (Section 2.1) we describe an analysis technique for Petri nets based on coverability graphs (Section 2.2). Section 2.3 describes an incremental coverability approach for Petri nets and is a result of this thesis. This technique will be used later (see Chapter 7.3) for the optimization of the unfolding procedure. Finally, in Section 2.4 we describe another approach to the analysis of Petri nets, namely, the backward coverability algorithm. Further analysis techniques for Petri nets are mentioned in Chapter 6 in the description of the verification tool AUGUR.

In this chapter we follow the presentation of [85] in Sections 2.1 and 2.2 and of [1] in Section 2.4.

2.1 Introduction to Petri Nets

Petri nets are a well-known formalism for the mathematical representation of distributed systems. The graphical structure of a Petri net is a directed bipartite graph with annotations. The states of distributed systems are represented by place nodes and the actions are represented by transition nodes. Places and transitions are connected via directed arcs (Fig. 2.1 (a)).

In order to define Petri nets and their markings, we first need the notion of multisets and ω -multisets.

Definition 2.1.1 ((ω -)multiset) *An ω -multiset over a set A is a mapping M from A to $\mathbb{N} \cup \{\omega\}$. It is called a (proper) multiset if the image of M does not contain ω . For an ω -multiset M we define $\Omega(M) = \{a \in A \mid M(a) = \omega\}$. The set of all ω -multisets of A is denoted by A_ω^\oplus , whereas the set of all multisets is denoted by A^\oplus .*

By \leq , \oplus , \ominus we denote the pointwise order (with ω as the largest element), addition and subtraction on multisets. Note that for $k, \ell \in \mathbb{N}$ it holds that $k - \ell = 0$ if $\ell > k$, $\omega + k = k + \omega = \omega + \omega = \omega$ and $\omega - k = \omega$. Furthermore $\omega - \omega$ is undefined, i.e., $-$ and \ominus are partial. Furthermore for $M \ominus M'$ we require that M' is a proper multiset.

A multiset $M \in A^\oplus$ can be written as a formal sum $M = \bigoplus_{a \in A} m_a \cdot a$ and given M we write $M(a)$ to denote the coefficient m_a .

A function $f: A \rightarrow B$ can be extended to a function $f: A_\omega^\oplus \rightarrow B_\omega^\oplus$ on ω -multisets as follows: For $M \in A_\omega^\oplus$ we define $M' = f(M)$ with $M'(b) = \sum_{a \in f^{-1}(b)} M(a)$ for every $b \in B$.

Now we are ready to define the notion of Petri net.

Definition 2.1.2 (Petri net) A (Petri) net is a tuple $\mathbf{N} = (S, T, \bullet(), ()^\bullet, m_0)$ where S is a set of places, T is a set of transitions, and $\bullet(), ()^\bullet: T \rightarrow S^\oplus$ assign to each transition its pre-set and post-set. Finally, $m_0 \in S^\oplus$ is the initial marking of \mathbf{N} .

An (ω) -marking of a Petri net \mathbf{N} is a multiset $m \in S^\oplus$ ($m \in S_\omega^\oplus$). Note that we will denote (ω) -markings with small letters, i.e., we write m for a marking instead of M , which we use for a multiset in general. We say that the Petri net has n tokens at place s in the marking m if $m(s) = n$.

In Fig. 2.1 (a) we represent a simple example of a Petri net having three places (s_1 , s_2 and s_3) and one transition t . Places s_1 and s_2 are in the pre-set of t with the values (called weights) 2 and 1 respectively. The place s_3 is in the post-set of t with the weight 1. The initial marking m_0 is defined as follows: $m_0(s_1) = 2$, $m_0(s_2) = 1$ and $m_0(s_3) = 0$. We depict tokens in the marking with small black circles.

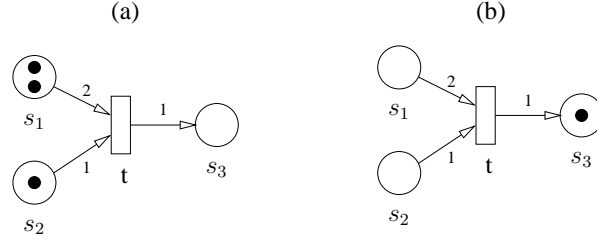


Figure 2.1: Example Petri net and a firing of transition

A transition $t \in T$ is *enabled* in a marking $m \in S_\omega^\oplus$ if $\bullet t \leq m$. If t is enabled in m then it can be *fired*. The firing of t removes from the current marking the pre-set and adds the post-set, resulting in a transition. If a transition t was fired in m then the obtained marking m' can be calculated as $m' = m \ominus \bullet t \oplus t^\bullet$. We write the firing of the transition t also in the form $m[t] m'$.

A marking m is said to be *reachable* in \mathbf{N} if there exists a sequence of markings $m_1, \dots, m_n = m$ and transitions t_1, \dots, t_n with $m_0[t_1] m_1 \dots m_{n-1}[t_n] m_n$. Furthermore a marking m' is called *coverable* if there exists a reachable marking m with $m' \leq m$.

In Fig. 2.1 (a), the transition t is enabled in the marking m_0 . The marking obtained after the firing of t is represented in Fig. 2.1 (b).

A Petri net \mathbf{N} is called *bounded* if there exists a number k such that for each reachable marking m and for each place s it holds that $m(s) \leq k$. Otherwise the Petri net is called *unbounded*. Being the over-approximations of infinite-state systems, the Petri nets considered in this paper are mostly unbounded. In the next sections of this chapter we show some analysis methods for unbounded Petri nets.

2.2 Coverability Graphs

Coverability graphs [85, 60] have proved to be a useful tool for the analysis of (unbounded) Petri nets. Also known under the names of Karp-Miller graphs or coverability trees, they can be used to gain full knowledge about the coverability of markings of a given net. That is, they answer the question whether a given marking m is coverable, i.e., whether

there exists a reachable marking m' with $m \leq m'$. Additionally they give information about the boundedness of a place and can be used for model-checking certain fragments of temporal logics [99].

While the reachability problem for Petri nets—although being decidable [80, 93]—is computationally infeasible for concrete applications due to the complexity of the algorithm, the coverability problem is still manageable for practical purposes, although it has a bad worst case behaviour (the size of the coverability graph may increase faster than any primitive recursive function [107]).

Coverability graphs can represent all reachable—or rather coverable—markings of a Petri net in the following way: nodes of the graph are ω -markings and for each reachable marking m in the net there exists an ω -marking m' in the graph that covers m . On the other hand, for each ω -marking in the graph there exists a reachable marking with the same number of tokens in all non- ω -places and more than i tokens in the ω -places for an arbitrary i . Edges are labelled with elements of T and represent the firing of transitions.

We will first define the notion of coverability graphs needed for this paper and then specify when a given coverability graph is valid for a net \mathbf{N} .

Definition 2.2.1 (Coverability graph) *A T -labelled (coverability) graph over S is a pair $G = (H, P)$, where $H \subseteq S_\omega^\oplus$ is a set of nodes and $P \subseteq H \times T \times H$ is a set of labelled edges.*

A path in a coverability graph is of the form $E_0 \xrightarrow{t_1} E_1 \dots E_{n-1} \xrightarrow{t_n} E_n$, where $(E_{i-1}, t_i, E_i) \in P$ for $i = 1, \dots, n$.

We need the definition of an i -marking which allows us to describe the meaning of omegas in the markings.

Definition 2.2.2 *Let E be a node in the coverability graph G and $\Omega(E) = \{s \in S \mid E(s) = \omega\}$. A marking m of \mathbf{N} is called an i -marking of E if for all $s \in \Omega(E)$, $m(s) \geq i$ and for all $s \notin \Omega(E)$, $m(s) = E(s)$.*

Now we are ready to define the notion of a valid coverability graph. Such graphs describe in a rather natural way all reachable and coverable markings in the given Petri net.

Definition 2.2.3 (Valid coverability graph) *Let $G = (H, P)$ be a coverability graph. A coverability graph is valid for a net $\mathbf{N} = (S, T, \bullet(), ()^\bullet, m_0)$ whenever the following holds:*

1. *For each firing sequence $m_0[t_1]m_1[t_2] \dots m_{n-1}[t_n]m_n$ there exists a path $E_0 \xrightarrow{t_1} E_1 \dots E_{n-1} \xrightarrow{t_n} E_n$ such that $m_0 = E_0$ and $m_i \leq E_i$ for all $i \in \{1, \dots, n\}$.*
2. *For every node $E \in H$ and every $i \in \mathbb{N}$ there exists a reachable marking m of \mathbf{N} which is an i -marking of E .*

Note that from 1 we can infer directly that for every reachable marking m of \mathbf{N} there exists a node $E \in H$ such that $m \leq E$.

A coverability graph represents all reachable markings of (finite) Petri nets either explicitly by some node or covered by some node.

Now we give construction for the coverability graph of a given Petri net \mathbf{N} . It is shown in [85] that this construction leads indeed to a valid coverability graph.

Construction 2.2.4 (Coverability graph)

Input: A Petri net $\mathbf{N} = (S, T, \bullet(), ()^\bullet, m_0)$.

Output: A valid coverability graph $G = (H, P)$ for \mathbf{N} .

Let $\Gamma = G_0, G_1, \dots$ be a sequence of graphs such that

1. $G_0 = (\{m_0\}, \emptyset)$.
2. Let $G_i = (H, P)$, $E \in H$ and $t \in T$ such that:
 - (a) t is enabled in E .
 - (b) no t -labelled edge (E, t, E') exists in P .

Then define the marking E' in the following way:

- (a) $E'(s) = \omega$ if there exists a node E'' in H such that $E'' \leq E \ominus \bullet t \oplus t^\bullet$ and $E''(s) < E(s) - \bullet t(s) + t^\bullet(s)$, and there exists a path from E'' to E in G_i .
- (b) $E'(s) = E(s) - \bullet t(s) + t^\bullet(s)$ otherwise.

The successor graph is $G_{i+1} = (H \cup \{E'\}, P \cup \{(E, t, E')\})$.

3. If it is not possible to construct G_{i+1} according to 2 then let $G_{i+1} = G_i$.

Γ is called a covering sequence and the coverability graph generated by Γ is

$$G = \left(\bigcup_{i=0}^{\infty} H_i, \bigcup_{i=0}^{\infty} P_i \right)$$

If the node E' is already contained in H then only a new arc (E, t, E') is added.

In Fig. 2.2 we represent a simple Petri net and its coverability graph. The coverability graph contains two nodes: the first node describes the initial marking m_0 and the second node is obtained after the firing of t . The numbers in the nodes for each place s correspond to $m(s)$. As we see in the second node we obtain an omega for the place s_2 because the node E' obtained after the first firing of t is strictly larger at the place s_2 than the initial marking E_0 , whereas $E_0 \leq E'$ and there exists a path from E_0 to E' .

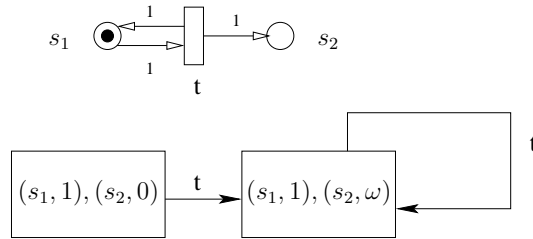


Figure 2.2: Petri net and a corresponding coverability graph

It is shown in [85] that due to Dixon's lemma the sequence G_0, G_1, \dots always becomes stationary at some point and hence the construction always terminates and the constructed graph is finite. Note however that the construction is non-deterministic and does not produce a unique coverability graph.

With respect to the analysis of Petri nets we are interested not only in the question if the given marking m is coverable by some marking m' with $m' \geq m$. We also search for a firing sequence $m_0 [t_1] m_1 [t_2] \dots m_{n-1} [t_n] m_n = m'$ leading from the initial

marking m_0 to the reachable marking m' . This is an important task for counterexample-guided abstraction refinement (Chapter 5). Usually we consider such firing sequence as a counterexample to a property to be verified.

We describe below the construction of a firing sequence to a reachable marking m' covering the marking m by the given Petri net \mathbf{N} and the valid coverability graph G . For each coverable marking m there exists a node E_m in the coverability graph G such that $m \leq E_m$.

Let $\Omega^a(E) \subseteq \Omega(E)$ be the set of places where a new ω -value has been produced during the construction of E in the coverability graph G . This set will be used below by the construction of firing sequence. We save for each node E and each $s \in \Omega^a(E)$ the sequence $\tau(E, s) = (E_0 \rightarrow_{t_1} E_1 \rightarrow_{t_2} \dots \rightarrow_{t_n} E_n = E')$, which was used at the construction time in order to produce an ω at the place s . Here $E_0 \leq E'$ and $E_0(s) < E'(s)$.

If $m \leq E_m$ then we search for the firing sequence to an i -marking m' of E_m , where i can be chosen suitably large in order to cover the marking m . We do this with the help of a recursive function *getTrace* which takes as parameters the node $E_m \in H_{\mathbf{N}}$ covering the marking m and an arbitrary number $i \in \mathbb{N}$.

The function *getPath*($E \in H_{\mathbf{N}}$) assumes that $\Omega(E) = \emptyset$ and returns with the help of a standard search algorithm in a directed graph the list of transitions $L = (t_1, \dots, t_l)$ leading from the node corresponding to the initial marking m_0 to the node E . Note that the initial node E_0 in the coverability graph corresponding to the marking m_0 has no omegas.

The next function that we use is *getPathToOmega*($E \in H_{\mathbf{N}}$), which assumes that $\Omega(E) \neq \emptyset$ and finds a node E_0^ω with $\Omega^a(E_0^\omega) \neq \emptyset$ together with a path

$$L = E_0^\omega = E_0 \rightarrow_{t_1} E_1 \rightarrow_{t_2} \dots \rightarrow_{t_n} E_n = E,$$

such that $\Omega^a(E_i) = \emptyset$ for $i > 0$. This can also be done with the help of a standard search algorithm in a directed graph. The function returns a path L together with its length and a maximum z of tokens disappearing in a single step of L in a single place. Formally

$$z = \max\{\bullet t_j(s) - t_j^\bullet(s) \mid 1 \leq j \leq n \wedge s \in P \wedge t_j^\bullet(s) < \bullet t_j(s)\}.$$

Construction 2.2.5 (Firing Sequence)

Input: A Petri net $\mathbf{N} = (S, T, \bullet(), ()^\bullet, m_0)$, a valid coverability graph $G = (H, P)$ and a coverable marking m (i.e., $m \leq E$ for some $E \in H$).

Output: A firing sequence $m_0[t_1]m_1[t_2] \dots m_{n-1}[t_n]m_n = m'$ such that $m \leq m'$. (We will output it in the form of a transition list).

function *getTrace*($E \in H_{\mathbf{N}}$, $i \in \mathbb{N}$)

Local Variables: List of Transitions $L = ()$

1. If $\Omega(E) \neq \emptyset$ then
2. $(L, \text{length}, z) = \text{GetPathToOmega}(E)$
3. foreach $s \in \Omega^a(E_0^\omega)$
4. Consider $\tau(E_0^\omega, s) = (E_0 \rightarrow_{t_1} E_1 \rightarrow_{t_2} \dots \rightarrow_{t_n} E_n = E_0^\omega)$
5. $z' = \max\{\bullet t_j(s) - t_j^\bullet(s) \mid 1 \leq j \leq n \wedge s \in P \wedge t_j^\bullet(s) < \bullet t_j(s)\}$
6. $z = \max(z', z)$
7. $L = (i + \text{length} \cdot z) \cdot (t_1, \dots, t_n) + L$
Here $l \cdot (t_1, \dots, t_n)$ means the list (t_1, \dots, t_n) repeated l times.

8. $length = length + n$
9. $L = getTrace(E_0^\omega, i + length \cdot z) + L$
10. $return L$
11. *Else return $getPath(E)$*

The correctness of this approach is described in [85] (Lemma 5.3 (d) (proof)). We also use this technique later in the proof of Proposition 2.3.7. Here we only give some intuition concerning the calculation.

First we find the previous node E_0^ω , where at least one omega has been added during the construction of the coverability graph. This is done by the function *GetPathToOmega*. Then for each $s \in \Omega^a(E_0^\omega)$ we consider a path $\tau(E_0^\omega, s)$ which was saved at construction time. We consider also the maximum z of tokens disappearing in a single step of the constructed part of the path in a single place.

The idea then is to iterate through $\tau(E_0^\omega, s)$ in order to obtain the necessary number of tokens in place s . To get this number we take into consideration a path from the current node E_0^ω to the node covering m . During this path some of the obtained tokens in s can be lost. To prevent this we iterate through $\tau(E_0^\omega, s)$ $i + length \cdot z$ times instead of i , where $length \cdot z$ token can be at most lost.

The obtained paths for E_0^ω are composed and the function *getTrace* is then recursively called for E_0^ω . Note that in E_0^ω there can still be places s' such that $s' \notin \Omega^a(E_0^\omega)$ and $s' \in \Omega(E_0^\omega)$.

There are some optimizations which can be made for practical usage of the algorithm. For example instead of the number z we can use a function $z : S \rightarrow \mathbb{N}$, which calculates tokens lost in a single step individually for each place $s \in S$. We can also set an individual bound i_s for each place $s \in S$ instead of calculating a general i -marking.

2.3 Incremental Calculation of Coverability Graphs

In this section we study the problem of the incremental construction of coverability graphs, which—to the best of our knowledge—has not been studied so far. That is, given a net \mathbf{N} and its coverability graph and this net is updated to \mathbf{N}' by adding some transitions or by merging transitions and/or places (more generally, this update is specified by a Petri net morphism), is there a simple way to compute a coverability graph for \mathbf{N}' without starting all over again?

This problem is motivated by the over-approximation of GTSs by Petri nets for verification purposes (Section 3.2). But we believe that such an incremental technique can also be useful for other applications where Petri nets are updated either interactively or automatically. The experimental results (Section 7.3) show that the application of this theory to the construction of the over-approximations is in fact feasible. The results of this section are published in [69].

The problem for the practical application of the new incremental coverability graph for \mathbf{N}' is that it will usually be larger than the coverability graph computed for \mathbf{N}' in a non-incremental way. In order to avoid costly blowups of the coverability graph, we suggest in Section 7.3 to restart the construction from scratch after N steps. Empirically, we have determined good values for N .

Given a net \mathbf{N} and a valid coverability graph G for \mathbf{N} , we want to perform some updates on \mathbf{N} , resulting in \mathbf{N}' . Here we show how G can be updated as well, leading to a coverability graph G' valid for \mathbf{N}' .

Updates can be the adding and merging of places and transitions, which are both captured by the notion of Petri net morphism (cf. [113, 20]). An example of a Petri

net morphism merging two places is depicted in Fig. 2.3. We do not consider here the deletion of nodes or transitions.

Definition 2.3.1 (Petri net morphism)

Let $\mathbf{N} = (S, T, \bullet(), ()^\bullet, m_0)$ and $\mathbf{N}' = (S', T', \bullet(), ()^\bullet, m'_0)$ be two Petri nets. A mapping $\varphi : \mathbf{N} \rightarrow \mathbf{N}'$ consisting of $\varphi_S : S \rightarrow S'$ and $\varphi_T : T \rightarrow T'$ is a Petri net morphism if $\bullet\varphi_T(t) = \varphi_S(\bullet t)$ and $\varphi_T(t)^\bullet = \varphi_S(t^\bullet)$ for each transition $t \in T$ and furthermore $m'_0 = \varphi_S(m_0)$.

Note that in the definition above we use the extension of function φ_S to multisets (see Definition 2.1.1). In the following we will usually drop indexes and denote φ_S as well as φ_T by φ .

Following Lemma describes some properties of the Petri net morphism φ .

Lemma 2.3.2 *Let m_1, m_2 be two (ω) -markings. Then it holds that:*

1. *Whenever $m_1 \leq m_2$, then also $\varphi(m_1) \leq \varphi(m_2)$.*
2. *$\varphi(m_1 \oplus m_2) = \varphi(m_1) \oplus \varphi(m_2)$.*
3. *Whenever $m_1 \geq m_2$, then $\varphi(m_1 \ominus m_2) = \varphi(m_1) \ominus \varphi(m_2)$.*

Proof: Follows directly from the fact that φ is a function on places: $\varphi_S : S \rightarrow S'$. \square

We will now show how a valid coverability graph for a net \mathbf{N} can be converted into a valid coverability graph for \mathbf{N}' .

Construction 2.3.3 (Incremental coverability graph)

Input: A Petri net morphism $\varphi : \mathbf{N} \rightarrow \mathbf{N}'$ and a valid coverability graph $G_{\mathbf{N}} = (H_{\mathbf{N}}, P_{\mathbf{N}})$ of \mathbf{N} .

Output: A valid coverability graph $G' = (H', P')$ for the Petri net \mathbf{N}' .

1. We first set

$$\hat{H} = \{\varphi(E) \mid E \in H_{\mathbf{N}}\}$$

and

$$\hat{P} = \{(\varphi(E_1), \varphi(t), \varphi(E_2)) \mid (E_1, t, E_2) \in P_{\mathbf{N}}\}.$$

That is, the markings of $G_{\mathbf{N}}$ are merged and the transitions are renamed according to the morphism φ .

2. Restart Construction 2.2.4 in Step 2 for \mathbf{N}' by considering (\hat{H}, \hat{P}) as a partially constructed coverability graph G_i . When it terminates we obtain a so-called incremental coverability graph.

Note that it could be the case that $\varphi(E_1) = \varphi(E_2)$ even if $E_1 \neq E_2$. The construction above ensures automatically that in this case the two corresponding nodes are merged. However, in an implementation care has to be taken to detect identical nodes and to merge them.

We will now show that the newly obtained incremental coverability graph is indeed valid for \mathbf{N}' and we start with the following lemmas.

Lemma 2.3.4 *Let $G_{\mathbf{N}'}$ be an incremental coverability graph and let (E'_1, t', E'_2) be an edge of the graph. Then*

1. $E'_1 \geq \bullet t'$

$$2. \forall s' \notin \Omega(E'_2): E'_2(s') = E'_1(s') - \bullet t'(s') + t' \bullet(s')$$

Proof: Straightforward from the construction. We only remark here that $-\bullet t, +t \bullet$ are monotone on markings. \square

Lemma 2.3.5 *Let $\varphi : \mathbf{N} \rightarrow \mathbf{N}'$ be a Petri net morphism and let $m_1 [t] m_2$ be a firing step in \mathbf{N} . Then there exists a firing step $\varphi(m_1) [\varphi(t)] \varphi(m_2)$ of \mathbf{N}' .*

Note that this implies immediately that $\varphi(\text{Reach}_{\mathbf{N}}) \subseteq \text{Reach}_{\mathbf{N}'}$, where $\text{Reach}_{\mathbf{N}}$ is the set of reachable markings of \mathbf{N} .

Proof: We mainly use Lemma 2.3.2: Since $\bullet t \leq m_1$, it follows that $\bullet \varphi(t) = \varphi(\bullet t) \leq \varphi(m_1)$. Hence $\varphi(t)$ is enabled in $\varphi(m_1)$. And we have $\varphi(m_1) \ominus \bullet \varphi(t) \oplus \varphi(t) \bullet = \varphi(m_1 \ominus \bullet t \oplus t \bullet) = \varphi(m_2)$. Hence $\varphi(m_1) [\varphi(t)] \varphi(m_2)$. \square

We first show that the newly constructed coverability graph is indeed valid for the net \mathbf{N}' .

Proposition 2.3.6 *Let $G_{\mathbf{N}'} = (H_{\mathbf{N}'}, P_{\mathbf{N}'})$ be the incremental coverability graph of \mathbf{N}' obtained according to Construction 2.3.3. Then $G_{\mathbf{N}'}$ satisfies Condition 1 of Definition 2.2.3.*

Proof: Let $m'_0 [t'_1] m'_1 [t'_2] \dots m'_{n-1} [t'_n] m'_n$ be a firing sequence of \mathbf{N}' with $m'_0 = \varphi(m_0)$. We now show the existence of a path $E'_0 \xrightarrow{t'_1} E'_1 \dots E'_{n-1} \xrightarrow{t'_n} E'_n$ with $m'_0 = E'_0$ and $m'_i \leq E'_i$ for all $i \in \{1, \dots, n\}$ by induction on n .

Since the original coverability graph $G_{\mathbf{N}}$ contained a node $E_0 = m_0$, the updated coverability graph will have a node $E'_0 = \varphi(E_0) = \varphi(m_0) = m'_0$ by construction. Now assume that the property has been shown for $n-1$, which implies that $m'_{n-1} \leq E'_{n-1}$. Since $\bullet t'_n \leq m'_{n-1}$ it follows that t'_n is enabled in E'_{n-1} and the construction must have ensured the existence of an edge (E'_{n-1}, t'_n, E'_n) .

If this edge has been created in Step 2 of the construction, we obtain

$$m'_n = m'_{n-1} \ominus \bullet t'_n \oplus t'_n \bullet \leq E'_{n-1} \ominus \bullet t'_n \oplus t'_n \bullet \leq E'_n.$$

Otherwise the edge has been created in Step 1 and hence there exists a transition t_n of \mathbf{N} with $\varphi(t_n) = t'_n$ and $E'_{n-1} = \varphi(E_{n-1})$, $E'_n = \varphi(E_n)$ for an edge (E_{n-1}, t_n, E_n) of the original coverability graph. In this case we have

$$\begin{aligned} m'_n &= m'_{n-1} \ominus \bullet t'_n \oplus t'_n \bullet \leq E'_{n-1} \ominus \bullet t'_n \oplus t'_n \bullet = \\ &\varphi(E_{n-1}) \ominus \varphi(\bullet t_n) \oplus \varphi(t_n \bullet) = \varphi(E_{n-1} \ominus \bullet t_n \oplus t_n \bullet) \leq \varphi(E_n) = E'_n, \end{aligned}$$

using Lemma 2.3.2. \square

Next we prove that the new coverability graph is not only an over-approximation, but also each node is a representative of a set of reachable markings, the i -markings of the node, where each ω -entry represents an unbounded place.¹ We show that each node of a valid coverability graph represents some set of reachable markings (and each ω -entry represents an unbounded place).

Proposition 2.3.7 *Let $G_{\mathbf{N}'} = (H_{\mathbf{N}'}, P_{\mathbf{N}'})$ be the incremental coverability graph of \mathbf{N}' obtained according to Construction 2.3.3. Then $G_{\mathbf{N}'}$ satisfies Condition 2 of Definition 2.2.3.*

¹Note that the proofs of the following propositions were obtained by adapting proofs from [85].

Proof: We will show that for each node E' and for each $i \in \mathbb{N}$ there exists a reachable i -marking m' of E' . The node E' was either obtained in Step 1 of the construction (merging) or in Step 2 (standard coverability algorithm).

In the first case we have that $E' = \varphi(E)$ for some node E of the original coverability graph. Hence there is a reachable i -marking of E in \mathbf{N} , which we denote by m . Now set $m' = \varphi(m)$ and check that m' is an i -marking of E' . Let s' be a place of \mathbf{N}' . Whenever $m'(s') \neq \omega$, then $m(s) \neq \omega$ for every $s \in \varphi^{-1}(s')$. So in this case we have

$$m'(s') = \sum_{s \in \varphi^{-1}(s')} m(s) = \sum_{s \in \varphi^{-1}(s')} E(s) = E'(s').$$

If however $m'(s') = \omega$, then $m(s) = \omega$ for at least one $s \in \varphi^{-1}(s')$. Here we have $m'(s') \geq m(s) \geq i$. It follows directly from Lemma 2.3.5 that m' is reachable.

In the second case E' has been added as described in Construction 2.2.4. We proceed by induction on the sequence of coverability graphs G_0, G_1, \dots constructed during the algorithm, where G_0 is the graph obtained by Step 1 of Construction 2.3.3. Now let (E, t, E') be a new edge (note that E, E' now denote different entities than above), and it follows from the induction hypothesis that E has an i -marking for every i . Furthermore we set $\hat{E} = E \ominus \bullet t \oplus t \bullet$ and we have $\hat{E} \leq E'$ and additionally $\Omega(E) \subseteq \Omega(E')$.

Now we show—by induction on the size of P —that for every $\Omega(E) \subseteq P \subseteq \Omega(E')$:

(*) for all $i \in \mathbb{N}$ there exists a reachable marking m' of \mathbf{N}' such that $\forall s' \in P : m'(s') \geq i$ and $\forall s' \notin \Omega(E') : m'(s') = E'(s')$.

For $P = \Omega(E)$ let $k = \max\{\bullet t(s') \mid s' \in S'\}$ be the maximal cardinality in $\bullet t$ and let m be an $(i + k)$ -marking of E . We set $m' = m \ominus \bullet t \oplus t \bullet$ and Condition (*) holds for m' since $m'(s') \geq i + m - \bullet t(s') \geq i$ for every $s' \in P$ and

$$m'(s') = m(s') - \bullet t(s') + t \bullet(s') = E(s') - \bullet t(s') + t \bullet(s') = \hat{E}(s') = E'(s')$$

for all $s' \notin \Omega(E')$.

Now assume some P with $\Omega(E) \subseteq P \subseteq \Omega(E')$ and let $p \in \Omega(E') \setminus P$. By construction there exists a node E_0 and a path $E_0 \xrightarrow{t_1} \dots \xrightarrow{t_n} E_n = E'$ such that $E_0 \leq \hat{E}$ and $E_0(p) < \hat{E}(p)$. To show (*) for $P \cup \{p\}$, we define z as the maximal number of tokens that is removed by any transition on the path.

$$z = \max\{\bullet t_j(s) - t_j \bullet(s) \mid 1 \leq j \leq n, s \in P, t_j \bullet(s) < \bullet t_j(s)\}$$

From the induction hypothesis it follows that there exists a reachable marking m_0 with $m_0(s) \geq i \cdot n \cdot z + i$ for all $s \in P$ and $m_0(s) = \hat{E}(s)$ for all $s \notin P$. From Lemma 2.3.4 it follows that starting from m_0 we can fire transitions t_1, \dots, t_n : $m_0[t_1] \dots [t_n] m_n$, and it holds that $m_n(s) \geq (i - 1) \cdot n \cdot z + i$ for all $s \in P$, $m_n(p) > m_0(p)$ and $m_n(s) = E'(s)$ for all $s \notin \Omega(E')$. So we can fire t_1, \dots, t_n even i times and the resulting marking m satisfies $m'(s) \geq i$ for all $s \in P$, $m'(p) \geq i$ and $m'(s) = m_0(s)$ for all $s \notin \Omega(E')$. This implies Property (*) for $P \cup \{p\}$. □

The propositions gives us the correctness of the incremental constructed coverability graph of \mathbf{N}' . It is now left to show that the construction of the incremental coverability graph terminates.

Proposition 2.3.8 *The incremental coverability graph $G_{\mathbf{N}'}$ is finite.*

Proof: Since the original coverability graph G_N is finite, the graph G_0 obtained after Step 1 is finite as well. In order to show that Step 2 terminates, we can adapt the standard termination proof for coverability graphs.

If the graph was not finite, then due to Koenig's lemma there exists an infinite acyclic path $K_0 \xrightarrow{t_1} K_1 \xrightarrow{t_2} \dots$. We consider only the suffix of the path that belongs to the newly constructed part of the coverability graph.

Due to Dixon's lemma, there must be two nodes K_i, K_j with $i < j$ and $K_i < K_j$. Then, because of the construction, at least one place becomes an ω place. Repeating this argument with the remaining path leads to the conclusion that the path must finally end in a node containing only ω 's, which is a contradiction to the fact that it is acyclic and infinite. \square

The incremental coverability graph $G_{N'}$ obtained from G_N may be larger than the coverability graph computed directly for N' . The reason for this is that we “forget” to introduce some of the ω 's, which would have been introduced in the coverability graph computed from scratch. These “forgotten” ω 's are then introduced at a later stage in the incremental construction, leading to a larger number of nodes.

Fig. 2.3 depicts such a situation. In the upper half there is a morphism between two Petri nets which merges places 1 and 2. Their corresponding coverability graphs are shown below as a) and b). Graph c) is the incremental coverability graph for the second net, obtained from a). As can be seen, graph c) is a valid coverability graph, but it has three nodes whereas graph b) has only two.

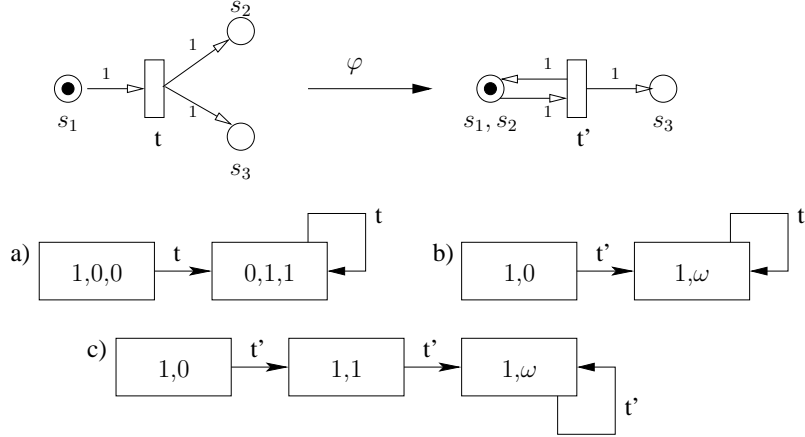


Figure 2.3: Possible increase in size for incremental coverability graphs.

This means that while small changes to a net affected by a morphism will usually lead to a coverability graph that is only moderately larger than the graph computed from scratch, large changes or several subsequent changes might significantly increase its size and lead to efficiency problems.²

In [44, 43] it has been suggested to find the minimal coverability graph, but this is a currently unsolved problem and would probably be quite costly even if solvable. So instead we propose to recompute the full coverability graph after several modifications of the net. In Section 7.3 we show how this technique can lead to an increase of efficiency in

²Naturally, the update itself could tremendously increase the size of the graph, for instance if it enables a transition that was not enabled before. But here we are rather interested in the ratio in size between the original graph and the incremental graph.

the case of the computation of approximated unfoldings of graph transformation systems, our application area. While this is quite a specific application, we believe that the considered Petri nets are of a rather arbitrary nature and there are no specific restrictions to the structure of nets that make the problem decidedly different from the general case.

2.4 Backward Coverability Algorithm

The construction of a coverability graph described above can be seen as a forward approach to the analysis of Petri nets. We have started with a single node representing an initial marking m_0 and have then fired the possible transitions on the already constructed nodes (representing the sets of markings). This section describes another (backward) approach to the analysis of Petri nets. It is the reachability algorithm described in [1] in a general form for well-structured transition systems [45]. We describe this approach here in terms of the Petri net formalism (Petri nets are considered in [1] as one of the proper examples of well-structured transition systems). The comparison of forward and backward analysis approaches for Petri nets [106] has shown that the result strongly depends on a concrete architecture of a given Petri net. In this thesis we will use both forward and backward algorithms for the analysis of Petri nets.

Let $\mathbf{N} = (S, T, \bullet(), ()^\bullet, m_0)$ be a Petri net. We use a function *select* from the set of all markings of \mathbf{N} to the power-set $\mathcal{P}(T)$ of transitions. The purpose of this function is to calculate the possible backwards steps from a given marking. We first define it as $select(m) = T$ for each marking m . Later it will be shown how to change it in order to optimize the backward coverability algorithm.

Construction 2.4.1 (Backward coverability algorithm)

Input: A Petri net $\mathbf{N} = (S, T, \bullet(), ()^\bullet, m_0)$ and a finite set of markings \mathcal{M} .

Output: **true** if some marking $m \in \mathcal{M}$ is coverable in \mathbf{N} and **false** otherwise.

Local Variables: A set V of markings representing the visited markings and a working set W of markings yet to be investigated.

1. Let $W = \mathcal{M}$ and let $V = \emptyset$.
2. While $W \neq \emptyset$ repeat steps 3 – 7
3. Select and remove $m \in W$
4. If there is $m' \in V$ such that $m \geq m'$, goto 2
5. If $m \leq m_0$ then exit with the result **true**
6. Add the markings $\{m \ominus t^\bullet \oplus \bullet t \mid t \in select(m)\}$ to W
7. Add m to V
8. If W becomes empty, then exit with the result **false**

This algorithm selects repeatedly the markings from W . If the obtained marking removed from W covers some already visited marking, then it is redundant and can be discarded. If the obtained marking is covered by the initial marking, then we can reproduce the firing of transitions in order to obtain some marking m'' such that $m'' \geq m'$ for some $m' \in \mathcal{M}$. This means that the marking $m' \in \mathcal{M}$ is coverable and the algorithm terminates with the answer **true**. Line 6 computes the new markings to be investigated. This backward step is a key point of the algorithm and we illustrate it in Fig. 2.4. Here

for a given marking m we calculate the marking $m \ominus t^\bullet \oplus {}^\bullet t$, which is a minimal marking allowing one to obtain a marking covering m after firing of t .

Line 7 adds the marking m to the already visited markings. If we terminate the while-cycle with empty W , then we conclude that the markings from \mathcal{M} are not coverable. The backward coverability algorithm terminates because the order \leq on markings considered as an order in well-structured transition systems is a well-quasi-order [45]. Termination of the algorithm with **true** means that it is possible to reach from some marking m'_0 underlying the initial marking ($m'_0 \leq m_0$) a marking m' covering some marking from \mathcal{M} .

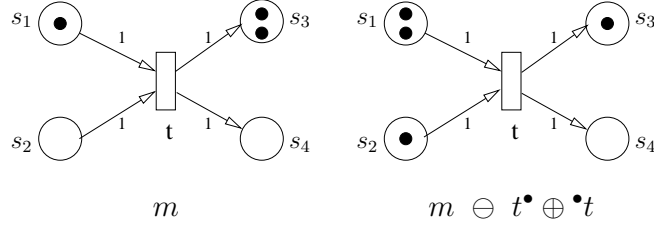


Figure 2.4: Calculation of a new marking in the backward coverability algorithm (step 6)

For analysis purposes (counterexample-guided abstraction refinement, Chapter 5) we need also a firing sequence (counterexample) leading from the initial marking m_0 to a reachable marking m'' such that $m'' \geq m'$ for $m' \in \mathcal{M}$. For this purpose we store for each marking m added at the step 6 of Construction 2.4.1 a sequence of transitions \mathcal{T}_m . We set $\mathcal{T}_{m_0} = \emptyset$ and if $m' = m \ominus t^\bullet \oplus {}^\bullet t$ then $\mathcal{T}_{m'} = t + \mathcal{T}_m$.

It is easy to see from Construction 2.4.1 that the sequence $\mathcal{T}_m = t_1, \dots, t_n$ obtained after the termination at line 5 of the construction corresponds to a firing sequence $m_0 [t_1 \rangle m_1 \dots m_{n-1} [t_n \rangle m_n = m''$ with $m'' \geq m'$ for $m' \in \mathcal{M}$.

We describe now how the function $select(m)$ can be changed in order to optimize the backward coverability algorithm (by reduction of the search space). The approach is based on the left-commutativity of transitions and is described in [1] for general well-structured transition systems. We describe it here only in its application to Petri nets.

We say that transitions t_1 and t_2 are *in conflict* if $t_1^\bullet \cap t_2^\bullet \neq \emptyset$. A transition t is *deficient* for a place s at the marking m if $m(s) < t^\bullet(s)$. We say that a transition t *separates* a marking m from a set M of markings if there is a place s with $t^\bullet(s) > 0$ such that $m(s) > m'(s)$ for each $m' \in M$.

The following algorithm proposed in [1] for well-structured transition systems calculates a modified version of $select(m)$, i.e., the algorithm generates a set of transitions to be explored from m .

Construction 2.4.2 ($select(m)$)

Input: A Petri net $\mathbf{N} = (S, T, \bullet(), ()^\bullet, m_0)$ and a marking m .

Output: A set of transition $T' \subseteq T$.

1. Start with some transition $select(m) = t_0$, which separates m from m_0 . If such a transition does not exist, then goto 3
2. Repeatedly add to $select(m)$ transitions t' for which there exists a transition $t \in select(m)$ such that one of the following conditions holds:
 - (a) t and t' are in conflict
 - (b) there exists a place s such that t is deficient for s at m and ${}^\bullet t'(s) > 0$

3. If no transition t_0 was found at the step 1 then let $select(m) = T$

We give now some intuition concerning the function $select(m)$. In the function we first take some transition t_0 separating m from m_0 in some place s . This means that probably we should fire t_0 on the path from m_0 to m . We also consider the case that some other transition t'_0 being in conflict with t_0 can be fired instead.

If some transition t was already included in $select(m)$ and t is deficient for some place s at m with $\bullet t'(s) > 0$, then it means that some tokens were put into s by firing t and taken away (probably) by t' . As in the previous step we take into consideration also all transitions being in conflict with t' . The function calculates $select(m)$ iteratively in this way.

Note that the backward coverability algorithm is still correct with this modified version of "select" (for more details see the presentation in [1]).

In the thesis we use both forward and backward analysis in order to analyse over-approximations of graph transformation systems (Chapter 3). Graph transformation systems are approximated by Petri graphs, which are Petri nets having additional hyper-graph structure.

In the verification of graph transformation systems coverability techniques can be used, for example, in the following way: Usually the property we want to verify is that some "bad" subgraph cannot be obtained during a system evaluation. If the marking corresponding to the subgraph is not coverable in the obtained Petri graph (in the underlying Petri net), then the subgraph cannot be a part of any reachable graph. In this case the property is shown to be true and the system is successfully verified.

In Section 6.3 more techniques will be introduced, which we use in practical analysis of the coverability problem in Petri nets.

Chapter 3

Graph Transformation Systems

In this chapter we introduce the class of (hyper)graph transformation systems (GTSs) considered in this thesis. Later in Chapter 4 this class of GTSs will be extended with attributes. Then we define the notion of Petri graph which is a Petri net with additional hypergraph structure, which will be used in order to approximate the behaviour of GTSs.

In the description of GTSs we focus on one of the most common approaches to graph rewriting, namely the DPO (double-pushout) approach, which derives its name from the fact that a rewriting step is described by two pushouts modelling the gluing of graphs. We are currently supporting restricted versions of DPO rules, where we only allow discrete interfaces, i.e., we can not describe preservation of edges, and merging as well as deletion of nodes is forbidden.

The theory of GTSs in this chapter is described on a rather concrete level. The justification of such an approach is the practical part of the thesis (Chapters 6 and 7), where all verification techniques described in this thesis are implemented in the frame of a new verification tool AUGUR 2.

We also describe an unfolding algorithm which computes the approximating Petri graph for a given GTS. Given a GTS, the algorithm produces a finite Petri graph such that every reachable in the GTS graph corresponds, in a sense formalized below, to a reachable marking in the Petri net underlying the obtained Petri graph.

For a given GTS an ordinary unfolding [11, 94] is constructed inductively, starting from the initial graph and then applying at each step all possible rules. An obtained result is acyclic and often infinite, also in the case of finite-state systems. To ensure that the algorithm produces a finite structure we consider – besides the unfolding rule, which extends the result – also a folding rule, which ”merges” some parts of the resulting graph.

In this chapter we follow the presentation of [6].

3.1 Introduction to Graph Transformation Systems

In this section we describe the notions of hypergraph and (hyper)graph transformation system (GTS).

For a set A we denote by A^* the set of strings over A and for a function $f : A \rightarrow B$ we denote by $f^* : A^* \rightarrow B^*$ its extension to strings.

We will in the following work with hypergraphs (also called graphs), a generalization of directed graphs, which are often more convenient for modelling.

Definition 3.1.1 (hypergraphs and hypergraph morphisms) *Let Λ be a set of labels where each label $l \in \Lambda$ has an arity $ar(l) \in \mathbb{N}$. A labelled hypergraph G is a*

tuple (V_G, E_G, c_G, l_G) , where V_G is a finite set of nodes, E_G is a finite set of edges, $c_G : E_G \rightarrow V_G^*$ is a connection function and $l_G : E_G \rightarrow \Lambda$ is the labeling function satisfying $\text{ar}(l_G(e)) = |c_G(e)|$ for every $e \in E_G$. The nodes are not labelled.

Let G and G' be two labelled hypergraphs. A hypergraph morphism (or simply morphism) $\varphi : G_1 \rightarrow G_2$ consists of a pair of total functions $\varphi_V : V_{G_1} \rightarrow V_{G_2}$ and $\varphi_E : E_{G_1} \rightarrow E_{G_2}$ such that for every $e \in E_{G_1}$ it holds that $l_{G_1}(e) = l_{G_2}(\varphi_E(e))$ and $\varphi_V^*(c_{G_1}(e)) = c_{G_2}(\varphi_E(e))$. A morphism is called *edge-bijective* (edge-injective) whenever it is bijective (injective) on edges. It is an *isomorphism* whenever it is bijective on nodes and edges.

Usually we are interested only in the structure of graphs, i.e., in graphs up to isomorphism. Furthermore we will in the following also abstract from isolated nodes, i.e., nodes not connected to any edge.

Hypergraphs can be rewritten using rules of the following kind.¹

Definition 3.1.2 (rewriting rule) A rewriting rule r is a triple (L, R, α) , where L and R are hypergraphs, called *left-hand side* and *right-hand side* respectively and $\alpha : V_L \rightarrow V_R$ is an injective mapping, indicating how nodes are preserved.

We demand that there are no isolated nodes in the left-hand side L and no isolated nodes in $V_R - \alpha(V_L)$. Additionally E_L must not be empty.

The first condition says that we abstract from isolated nodes, whereas the second is a standard requirement for unfolding-based techniques, where every rule must be consuming. Note furthermore that we do not consider rules that preserve edges of the left-hand side.

In the categorical DPO approach a rule is considered as span of injective hypergraph morphisms $L \leftarrow_{id} K \rightarrow_{\alpha} R$, where K is the discrete graph consisting of the nodes in V_L (see Appendix A for more details on the categorical approach to graph rewriting).

For convenience we will in the following often assume that α is an inclusion denoted by id , which can be enforced by renaming the nodes of the left or right hand side appropriately, and that the node and edge sets of L and R are disjoint otherwise. That is, we demand that $V_L \subseteq V_R$ and $E_L \cap E_R = \emptyset$, which implies that the union $L \cup R$ is well-defined.

Given a hypergraph, a rewriting rule and a match of the left-hand side, we can apply this rule and replace the left-hand side by the right-hand side in the following way. Additionally we define a partial morphism ν from the original graph to the rewritten graph, keeping track of preserved nodes and edges.

Definition 3.1.3 (rewriting step) Let $r = (L, R, id)$ be a rewriting rule. A match of r in a hypergraph G is any morphism $\varphi : L \rightarrow G$ injective on edges. We can apply r to G according to the match φ and obtain a new graph H , written $G \Rightarrow_r H$, which is defined as follows: $[V_H = V_G \uplus (V_R - V_L) \quad E_H = (E_G - \varphi(E_L)) \uplus E_R]$ and, defining $\bar{\varphi} : V_R \rightarrow V_H$ by $\bar{\varphi}(v) = \varphi(v)$ if $v \in V_L$ and $\bar{\varphi}(v) = v$ otherwise, the connection and labelling functions are given by

$$\begin{aligned} e \in E_G - \varphi(E_L) &\Rightarrow c_H(e) = c_G(e), \quad l_H(e) = l_G(e) \\ e \in E_R &\Rightarrow c_H(e) = \bar{\varphi}^*(c_R(e)), \quad l_H(e) = l_R(e) \end{aligned}$$

We also define an injective partial morphism $\nu : G \rightarrow H$ where $\nu_V : V_G \rightarrow V_H$ and $\nu_E : (E_G - \varphi(E_L)) \rightarrow E_H$ with $\nu(x) = x$ for every node or edge x .

¹Although our rewriting rules can be seen within the framework of the DPO approach [95], we are using simpler rules with only discrete interfaces, where additionally the deletion of nodes is forbidden.

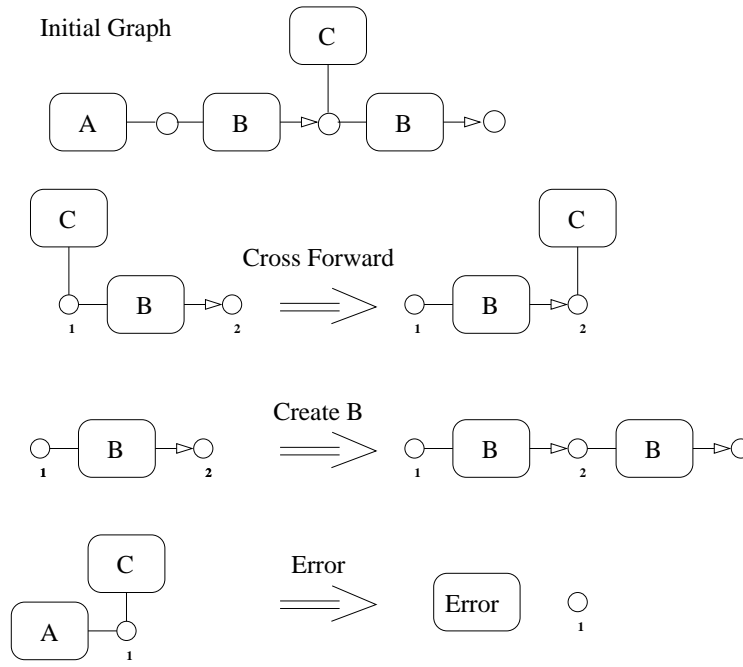


Figure 3.1: Example of GTS

We are now ready to define the notion of graph transformation system.

Definition 3.1.4 (graph transformation system) A graph transformation system (GTS) $\mathcal{G} = (\mathcal{R}, G_0)$ is a finite set of rules together with a start hypergraph (also called initial graph).

We illustrate the definitions of this chapter with a simple example in Fig. 3.1. This example GTS consists of an initial graph and of three rules: "Cross Forward", "Create B" and "Error". The system contains edges of three types labelled "A", "B" and "C". The edge "C" can cross the edge "B", but only in one direction (Rule: Cross Forward). From a single edge with the label "B" a second edge labelled "B" can be created (Rule: Create B). This is an infinite state system because an infinite number of edges labelled "B" can be created. The property we want to verify is that the edges "A" and "C" will never be connected. If this situation is detected, rule "Error" will be applied and an 0-ary edge labelled *Error* is created.

3.2 Verification of Graph Transformation Systems

The verification of GTSs is undecidable in general because GTSs are Turing-powerful. In the last few years a technique for analysing of graph transformation systems based on approximations has been developed [12, 6]. In order to approximate GTSs, Petri nets are employed, which, as multiset rewriting systems, can be seen as a special case of graph rewriting. Petri nets are an easier model than GTSs and hence more amenable to analysis; several algorithms and tools are available for their verification. Furthermore, by approximating with Petri nets we will be able to preserve nice properties of the GTS model, such as locality (state changes are only described locally) and concurrency (no unnecessary interleaving of events) in the approximation.

In this section we will give an overview over a technique that approximates a graph transformation system by a structure that is both a Petri net and a hypergraph [6, 4, 13].

First we define the notion of Petri graph, which will be used to represent an over-approximation for a given GTS. Note that the edges of the graph are at the same time the places of the net and that the transitions are labelled with rules of the GTS.

Definition 3.2.1 (Petri graph) *Let $\mathcal{G} = (\mathcal{R}, G_0)$ be a GTS. A Petri graph (over \mathcal{R}) is a tuple $P = (G, N, \mu)$, where G is a hypergraph, $N = (E_G, T_N, \bullet(), ()^\bullet, p_N)$ is an \mathcal{R} -labelled Petri net (initial marking m_0 is defined below) where the places are the edges of G and μ associates to each transition $t \in T_N$, with $p_N(t) = (L, R, id)$, a hypergraph morphism $\mu(t) : L \cup R \rightarrow G$ such that $\bullet t = \mu(t)^\oplus(E_L)$ and $t^\bullet = \mu(t)^\oplus(E_R)$.*

A Petri graph for the GTS \mathcal{G} is a pair (P, ι) , where $P = (G, N, \mu)$ is a Petri graph over \mathcal{R} and $\iota : G_0 \rightarrow G$ is a graph morphism. A marking is reachable (coverable) in a Petri graph if it is reachable (coverable) in the underlying Petri net with the multiset $\iota^\oplus(E_{G_0})$ as the initial marking m_0 .

We view Petri graphs as symbolic representations of transition systems with graphs as states. Specifically, each marking m of a Petri graph (G, N, μ) can be seen as representation of a graph, denoted by $graph(m)$, according to the following definition: We take the marked subgraph of G and duplicate each edge as indicated by the marking.

Definition 3.2.2 (graph generated by a marking) *Let $P = (G, N, \mu)$ be a Petri graph and let $m \in E_G^\oplus$ be a marking of N . The graph generated by m , denoted by $graph_G(m)$ or $graph(m)$, is the graph H defined as follows:*

$$V_H = \{v \in V_G \mid \exists e \in m \exists i : (c_G)_i(e) = v\},$$

$$E_H = \{(e, i) \mid e \in m \wedge 1 \leq i \leq m(e)\},$$

$$c_H((e, i)) = c_G(e) \quad \text{and} \quad l_H((e, i)) = l_G(e).$$

Note that by $(c_G)_i(e)$ we denote the i -th node in the sequence $c_G(e)$.

Alternatively one can define $graph(m)$ as the unique graph H , up to isomorphism, such that H has no isolated nodes and there exists a morphism $\psi : H \rightarrow G$ injective on nodes with $\psi^\oplus(E_H) = m$. Furthermore, whenever there exists a morphism $\varphi : G' \rightarrow G$ such that $\varphi^\oplus(E_{G'}) \leq m$, then there exists an edge-injective morphism $e_{m, \varphi} : G' \rightarrow graph(m)$ such that $\psi \circ e_{m, \varphi} = \varphi$. This morphism $e_{m, \varphi}$ will be used later in the paper.

This morphism $e_{m, \varphi}$ is not unique, since we may have several parallel edges from which an image can be chosen, but the resulting diagram consisting of $e_{m, \varphi}, \varphi, \psi$ is unique up to isomorphism.

Every hypergraph G can be considered as a Petri graph (G, N, μ) over \mathcal{R} by taking N as a net with $S_N = E_G$ and no transitions. Similarly G_0 can be seen as Petri graph for (\mathcal{R}, G_0) by taking as $\iota : G_0 \rightarrow G_0$ the identity.

We now introduce a merging operation on Petri graphs which constructs the factoring of Petri graphs through an equivalence induced by a suitable relation.

Definition 3.2.3 (consistent relation on a Petri graph) *Let $P = (G, N, \mu)$ be a Petri graph and let \equiv be a relation on $V_G \cup E_G \cup T_N$ (assume the sets V_G, E_G, T_N are disjoint). We say that \equiv is consistent when*

1. if $x \equiv x'$ then $x, x' \in X$ for some $X \in \{V_G, E_G, T_N\}$
2. for all $e, e' \in E_G$ if $e \equiv e'$ then $l_G(e) = l_G(e')$

3. for all $t, t' \in T_N$ if $t \equiv t'$ then $p_N(t) = p_N(t')$

Definition 3.2.4 (closed relation on a Petri graph) A consistent relation \equiv over P is called closed if for all $t, t' \in T_N$, $e, e' \in E_G$ the following properties hold

1. $p_N(t) = p_N(t') = (L, R, \alpha) \wedge (\forall e \in E_L : \mu(t)(e) \equiv \mu(t')(e)) \Rightarrow t \equiv t'$
2. $t \equiv t' \Rightarrow \forall e \in E_L \cup E_R : \mu(t)(e) \equiv \mu(t')(e)$
3. $e \equiv e' \wedge c_G(e) = v_1 \dots v_m \wedge c_G(e') = v'_1 \dots v'_m \Rightarrow \forall 1 \leq i \leq m : v_i \equiv v'_i$

To ensure that the factoring of a Petri graph with respect to a relation is well-defined, the relation must be closed. The following observation is important for defining the merging operation.

Fact. Any consistent relation over a Petri graph P can be extended to a least closed equivalence relation.

Definition 3.2.5 (Petri graph merging) Let $P = (G, N, \mu)$ be a Petri graph and let \equiv be a consistent relation over P . Then the merging of P with respect to \equiv denoted by P/\equiv , is the Petri graph (G', N', μ') defined as follows. First we extend \equiv to the least closed equivalence relation. Then we put

$$G' = (V_{G/\equiv}, E_{G/\equiv}, c_{G'}, l_{G'}),$$

where $c_{G'}([e]_{\equiv}) = [v_1]_{\equiv} \dots [v_n]_{\equiv}$ and $l_{G'}([e]_{\equiv}) = l_G(e)$ whenever $e \in E_G$ and $c_G(e) = v_1 \dots v_n$. Furthermore

$$N' = (E_{G'}, T_{N'/\equiv}, \bullet(), ()^\bullet, p_{N'}),$$

where $\bullet[t]_{\equiv} = \bigoplus_{e \in \bullet t} [e]_{\equiv}$, $[t]_{\equiv}^\bullet = \bigoplus_{e \in t^\bullet} [e]_{\equiv}$ and $p_{N'}([t]_{\equiv}) = p_N(t)$ whenever $t \in T_N$.

For each $t \in T_N$ the morphism $\mu'([t]_{\equiv})$ is defined by $\mu'([t]_{\equiv})(x) = [\mu(t)(x)]_{\equiv}$ for any graph item x in the rule $p_N(t)$.

In order to obtain a Petri graph approximating a GTS, we first need—as building blocks—Petri graphs that describe the effect of a single rule.

Definition 3.2.6 (Petri graph for a rewriting rule) Let $r = (L, R, id)$ be a rewriting rule. By $P(t, r) = (G, N, \mu)$ we denote a Petri graph with $G = L \cup R$ and N a net with places $S_N = E_L \cup E_R$ and one transition t such that $p_N(t) = r$, $\bullet t = E_L$ and $t^\bullet = E_R$. Furthermore the morphism $\mu(t): L \cup R \rightarrow G$ is the identity.

Given a GTS $\mathcal{G} = (\mathcal{R}, G_0)$, one can construct an over-approximating Petri graph $\mathcal{C}_{\mathcal{G}}$ (also called the *covering* of \mathcal{G}), using the following algorithm (see [6]). It starts with a Petri graph P_0 that consists only of the start graph and computes $\mathcal{C}_{\mathcal{G}}$ iteratively. It is based on an unfolding technique which is combined with over-approximating folding steps which guarantee a finite approximation.

Construction 3.2.7 (approximated unfolding) We set $P_0 = (G_0, N_0, m_0)$, where N_0 contains no transitions, $m_0 = E_{G_0}$ and let $\iota_0: G_0 \rightarrow G_0$ be the identity. As long as one of the following steps is applicable, transform P_i into P_{i+1} according to the possibilities given below (where folding steps take precedence over unfolding steps).

Unfolding: Find a rule $r = (L, R, id) \in \mathcal{R}$ and a match $\varphi: L \rightarrow G_i$ that has not yet been unfolded. Then choose a new transition t and extend P_i by attaching $P(t, r)$, i.e., take

the disjoint union of both Petri graphs and factor through the equivalence \equiv generated by $e \equiv \varphi(e)$ for every $e \in E_L$:

$$\text{unf}((P, \iota), r, \varphi) = (\{P, P(t, r)\} / \equiv, \iota / \equiv).$$

A categorical description of the unfolding operation as pushout can be found in Appendix A.

Folding: Find a rule $r = (L, R, id) \in \mathcal{R}$ and two matches $\varphi, \varphi' : L \rightarrow G_i$ such that the second match is causally dependent on the transition unfolding the first match. Then merge the two matches by setting $\varphi(e) \equiv \varphi'(e)$ for each $e \in E_L$ and factoring through the resulting equivalence relation \equiv :

$$\text{fold}((P, \iota), r, \varphi', \varphi) = (P / \equiv, \iota / \equiv).$$

A categorical description of the folding operation as coequalizer can be found in Appendix A.

If neither possibility applies the Petri graph P_i obtained in the last step is returned. The result is denoted by C_G .

In the more exact version of the algorithm [6], a left-hand side should only be unfolded or folded only if it is coverable in the net underlying the Petri graph. This means if $\varphi^\oplus(E_L)$ is a coverable marking in P_i . This condition can be relaxed, which however leads to less precise approximations.

Example: We illustrate the algorithm using the rules of the example in Fig. 3.1 by starting with an initial graph and making first two steps of the approximated unfolding algorithm Fig. 3.2. First we put an initial marking in the edges of the initial graph. In the first step the rule "Cross Forward" is applied in the unfolding step. The second is a folding step which is applied for the rule "Cross Forward". With this step two edges labelled with "B" are merged. Note that transitions modelling the consumption and production of tokens can be seen as specifying the deletion and creation of edges.

The constructed over-approximation for the GTS in Fig. 3.1 is represented in Fig. 3.3. As we can see the "Error" rule can be applied and the "Error" edge will be created. This means that either the property we want to verify does not hold or the obtained over-approximation is too coarse. For this example the latter is the case and we will show in the following how the Petri graph can be refined in order to verify the system.

Now we introduce the notion of a *depth of items* in a Petri graph. We will use this notion below in this chapter in order to represent a simple refinement technique for Petri graphs approximating GTSs. In order to deal with causal cycles we use the set of natural numbers extended with "infinity".

Definition 3.2.8 We denote by \mathbb{N}^ω the partially ordered set $(\mathbb{N} \cup \{\omega\}, \leq)$, where \leq is the usual order on natural numbers and $n \leq \omega$ for all $n \in \mathbb{N}$.

Addition can be extended to \mathbb{N}^ω in the following way: $m + n$ is a standard addition if $m, n \in \mathbb{N}$ and $m + n = \omega$ otherwise. We start with a definition of depth for Petri nets and extend it later to Petri graphs.

Definition 3.2.9 (Depth in a Petri net) Let $\mathbf{N} = (S, T, \bullet(), ()^\bullet, M_0)$ be a Petri net. By $\sqcup X$ we denote the least upper bound of the set X . We define a function

$$D : (S \cup T \rightarrow \mathbb{N}^\omega) \rightarrow (S \cup T \rightarrow \mathbb{N}^\omega)$$

as follows:

$$\begin{aligned} D(d)(x) &= \sqcup \{d(t) \mid t \in T \wedge x \in t^\bullet\} \text{ if } x \in S \text{ and} \\ D(d)(x) &= \sqcup \{d(s) \mid s \in S \wedge s \in \bullet x\} + 1 \text{ if } x \in T. \end{aligned}$$

Then the function $\text{depth} : S \cup T \rightarrow \mathbb{N}^\omega$ that assigns depth information to every item is $\text{fix}(D)$, i.e., the least fixed point of D .

Notice that $S \cup T \rightarrow \mathbb{N}^\omega$ with the point-wise order is a complete partial order and D is monotone and continuous. Therefore the depth exists and is the least upper bound of the chain $(D^n(\mathbf{0}))_n$, where the constant function $\mathbf{0}$ maps every item to 0.

The function depth assigns to each item of a Petri net its causal depth. In particular an item x located in a causality cycle has an infinite depth, i.e., $\text{depth}(x) = \omega$.

Following definition extends the notion of depth to Petri graphs, where we additionally define it for nodes.

Definition 3.2.10 (Depth in a Petri graph) Let $P = (G, N, \mu)$ be a Petri graph. The function $\text{depth} : E_N \cup T_N \rightarrow \mathbb{N}^\omega$ is defined as in Definition 3.2.9 (here places of the Petri net are the edges of P). The extension of this function for nodes $v \in V$ is

$$\text{depth}(v) = \bigsqcup \{ \text{depth}(t) \mid v \in \mu(t)(V_R \setminus V_L) \}$$

The depth of a node v is the maximal depth of rules with left-hand side L and right-hand side R , where v "appears" in $V_R \setminus V_L$, i.e., intuitively, of rules which can "generate" v .

Now we are ready to define the notion of a k -covering. We call a Petri graph $\mathcal{C}_G^k = (P, \iota)$ k -covering of the given GTS \mathcal{G} if it is obtained according to Construction 3.2.7 with following changes in the folding steps:

Construction 3.2.11 (Folding step in a k -covering)

Folding: For a rule $r = (L, R, id) \in \mathcal{R}$ and two matches $\varphi, \varphi' : L \rightarrow G_i$ (such that $\varphi^\oplus(E_L)$ and $\varphi'^\oplus(E_L)$ are coverable in N_i and the second match is causally dependent on the transition t unfolding the first match) we additionally demand that

1. For each node $v \in V_L$ it holds that $\text{depth}(\varphi(v)) \geq k$ and $\text{depth}(\varphi'(v)) \geq k$
2. For each edge $e \in E_L$ it holds that $\text{depth}(\varphi(e)) \geq k$ and $\text{depth}(\varphi'(e)) \geq k$
3. $\text{depth}(t) \geq k$

Such two matches are merged by setting $\varphi(e) \equiv \varphi'(e)$ for each $e \in E_L$ and factoring through the resulting equivalence relation \equiv .

Approximating Petri graph \mathcal{C}_G obtained according to Construction 3.2.7 without the changes above can be seen as a 0-covering \mathcal{C}_G^0 of the given GTS \mathcal{G} . The following proposition [12] describes the termination result for the construction of a k -covering.

Proposition 3.2.12 (Termination) The algorithm computing the k -covering terminates for every GTS \mathcal{G} and every $k \in \mathbb{N}$.

The fact that the algorithm computing the k -covering always terminates is not obvious at first glance. The proposition means that given a GTS \mathcal{G} we can always construct its k -covering \mathcal{C}_G^k in a finite number of steps.

The following property of a k -covering that we want to describe here is a *confluence*. The confluence of the algorithm is not further used in this thesis and is given here only as an interesting and nice result about the approximation of GTSs by Petri graphs. We first need a notion of an *irredundant* Petri graph.

Definition 3.2.13 (Irredundancy Condition) A Petri graph P is called irredundant if whenever there are transitions t_1, t_2 having the same labels with $\bullet t_1 = \bullet t_2$, then it holds that $t_1 = t_2$.

The following proposition [12] holds only if we consider exclusively irredundant Petri graphs. Each Petri graph can be changed into an irredundant one by factoring through the equivalence relation obtained from $t_1 \equiv t_2$ for each t_1 and t_2 such that $\bullet(t_1) = \bullet(t_2)$, t_1 and t_2 have the same labels and $t_1 \neq t_2$. Specifically, whenever we lose irredundancy by folding, it can be enforced by merging transitions having the same pre-set.

Proposition 3.2.14 (Confluence) *For any input GTS \mathcal{G} and every $k \in \mathbb{N}$ the algorithm computing the k -covering terminates with a result $\mathcal{C}_{\mathcal{G}}^k$, unique up to isomorphism.*

Before we can show in what way Petri graphs can be considered as abstractions of GTSs and before we discuss how they can be analyzed, we first need the definition of an abstract run of a GTS and a notion of correspondence of two abstract runs. Then we can define how Petri graphs can be seen as abstractions of GTSs.

Definition 3.2.15 (Abstract run) *An abstract run of a GTS (\mathcal{R}, G_0) is a sequence of hypergraphs $\mathcal{J} = (J_0 \Rightarrow_{r_1} J_1 \Rightarrow_{r_2} \dots \Rightarrow_{r_n} J_n)$, where r_i is a rule name, together with morphisms $\varphi_i : L_{i+1} \rightarrow J_i$ for each $i = 1, \dots, n-1$, where L_i is the left-hand side of rule $r_i \in \mathcal{R}$.*

Note that we do not demand that J_i can be derived from J_{i-1} by applying rule r_i at match φ_i . If this is the case \mathcal{J} will be called a real run and we will also use the symbol \Rightarrow instead of \Rightarrow .

Let $\mathcal{J}' = (J'_0 \Rightarrow_{r_1} J'_1 \Rightarrow_{r_2} \dots \Rightarrow_{r_n} J'_n)$ be another abstract run with morphisms $\varphi'_i : L_{i+1} \rightarrow J'_i$ for each $i = 1, \dots, n-1$. We say that \mathcal{J}' weakly corresponds to \mathcal{J} (in symbols $\mathcal{J}' \ll \mathcal{J}$) if there exist edge-bijective morphisms $\xi_i : J'_i \rightarrow J_i$ for $i = 0, \dots, n$. If furthermore the following diagram commutes for $i = 0, \dots, n-1$ we say that \mathcal{J}' corresponds to \mathcal{J} and write $\mathcal{J}' \lll \mathcal{J}$.

$$\begin{array}{ccccc} L_{i+1} & \xrightarrow{\varphi'_i} & J'_i & \xrightarrow{\xi_i} & J_i \\ & \searrow \varphi_i & & \nearrow & \\ & & & & \end{array}$$

Note that the notion of correspondence considered here does not demand that the occurrences of right-hand sides coincide as well. This will also mean that more abstract runs correspond to a spurious run, which is an advantage since these are possibly eliminated by abstraction refinement.

Petri graphs can, as mentioned above, be seen as symbolic representations of graph transition systems and also as representations of sets of abstract runs.

Let us remind here that given a hypergraph $\text{graph}(m)$ and a morphism $\psi : \text{graph}(m) \rightarrow G$ injective on nodes with $\psi^\oplus(E_H) = m$ and a morphism $\varphi : G' \rightarrow G$ such that $\varphi^\oplus(E_{G'}) \leq m$ there exists an edge-injective morphism $e_{m,\varphi} : G' \rightarrow \text{graph}(m)$ such that $\psi \circ e_{m,\varphi} = \varphi$. The morphism $e_{m,\varphi}$ is used in the following definition. This morphism $e_{m,\varphi}$ is not unique, since we may have several parallel edges from which an image can be chosen, but the resulting diagram consisting of $e_{m,\varphi}, \varphi, \psi$ is unique up to isomorphism.

Definition 3.2.16 (Abstract runs of a Petri graph) *Let (P, ι) with $P = (G, N, \mu)$ be a Petri graph for a GTS (\mathcal{R}, G_0) . Furthermore let $m_0[t_1] \dots [t_n]m_n$ be a firing sequence of the net N and let $r_i = p_N(t_i)$ be the rules corresponding to the transitions. We define morphisms $\varphi_i = e_{m_i, \mu(t_{i+1})|_{L_{i+1}}} : L_{i+1} \rightarrow \text{graph}(m_i)$, where L_{i+1} is the left-hand side of rule r_{i+1} . The sequence $\text{graph}(m_0) \Rightarrow_{r_1} \text{graph}(m_1) \Rightarrow_{r_2} \dots \Rightarrow_{r_n} \text{graph}(m_n)$ together with the morphisms φ_i is an abstract run. We denote by $\text{Run}_A(P, \iota)$ the set of all abstract runs of the Petri graph (P, ι) .*

Each real run $\mathcal{J}_r = (G_0 \Rightarrow_{r_1} G_1 \Rightarrow_{r_2} \dots \Rightarrow_{r_n} G_n)$ of the GTS (\mathcal{R}, G_0) can be considered as an abstract run where the $\varphi_i : L_{i+1} \rightarrow G_i$ represent the matches of the left-hand sides of the rules r_i .

The following result [12] allows us to consider the obtained Petri graphs as over-approximations of the original GTS.

Proposition 3.2.17 *Let \mathcal{C}_G^k be a computed k -covering for a GTS \mathcal{G} and $k \in \mathbb{N}$. Then, for every real run \mathcal{J}_r of the graph transformation system there exists an abstract run $\mathcal{J} \in \text{Run}_A(\mathcal{C}_G^k)$ such that \mathcal{J}_r corresponds to \mathcal{J} , i.e., $\mathcal{J}_r \lll \mathcal{J}$.*

In Fig. 3.4 there is a real run \mathcal{J}_r of the GTS from Fig. 3.1 and the corresponding abstract run \mathcal{J} of a Petri graph from Fig. 3.3.

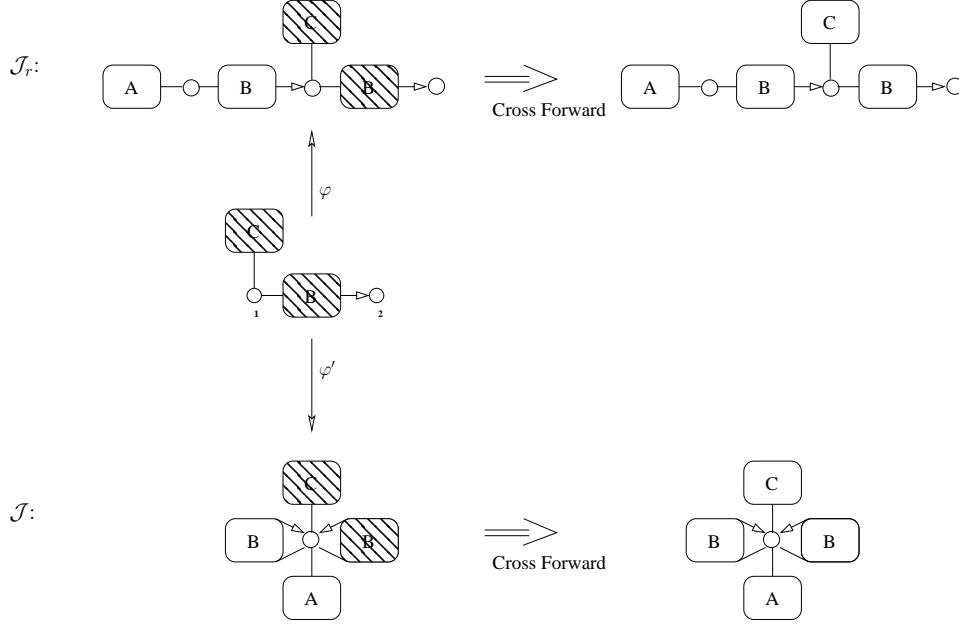


Figure 3.4: Real run and the corresponding abstract run of a Petri graph.

An abstract run \mathcal{J} for which there does not exist a real run corresponding to \mathcal{J} is called *spurious*. If, at the same time, it violates the property we attempt to verify, it is called a *counterexample* or *error trace*.

We can now verify GTSs by analyzing the Petri graph and the underlying Petri net. For instance, in order to show that no reachable graph contains a subgraph G_s we add a new rule to the GTS with G_s as left-hand side and an edge with a new label *Error* in the right-hand side (see rule “Error” in Fig. 3.1). If we can show that either no place labelled with “Error” exists in the net or every such place is not coverable, then we can deduce that this property holds. This can be shown using coverability graphs or the backward coverability algorithm described in Chapter 2.

However, if the approximation is too coarse, we might not be able to verify the property. The approximated unfolding can then be refined by forbidding certain folding steps. In the depth-based approach described above, we can forbid in the construction of a k -covering to merge items in the unfolding the depth of which is smaller than some fixed constant k . This eliminates all spurious runs of length smaller than k , but in the limit the k -coverings converge to the full unfolding, which is obtained by unfolding without folding steps and which is in general infinite.

The described depth-based approach to the refinement of the approximation does not take into account the property which should be checked and the subsequently better approximations grow in size fairly rapidly. For our example in Fig. 3.1 and $k = 1$ we obtain the Petri graph having 21 edges, 6 vertexes and 21 transitions (instead of 4 edges, one node and 3 transitions for $k = 0$). Although in this case the property can be successfully verified using the 1-covering, still the state space grows rather heavily (for such a simple example).

In Chapter 5 we represent another refinement approach (which is the main theoretical outcome of this thesis) showing how to successfully apply the technique of counterexample-guided abstraction refinement in the framework of GTSs (Section 5.1). For the example in Fig. 3.1 we obtain a refined Petri graph having only 4 edges, 2 vertexes and 3 transitions which also allows us to successfully verify the property.

Another comparison between the depth-based approach and the counterexample-guided abstraction refinement can be found in Chapter 7 (Case Studies), Sections 7.1 and 7.2.

Chapter 4

Attributed Petri Nets and Graph Transformation Systems

In this section we describe our view on attributed graph transformation systems (AGTSS) labelled over an algebra, and we approximate AGTSSs by attributed Petri graphs, which are basically coloured Petri nets equipped with a hypergraph structure. Our definition is closely related to the presentations in [58, 39].

After the usual approximation, attributes are added to the resulting Petri graph, which can then be seen and analyzed as a coloured Petri net. For the analysis of coloured Petri nets we use in this chapter adaptations of forward and backward coverability algorithms (Chapter 2).

Since the carrier sets of data types are often infinite, we need also attribute abstraction. This is standard in the framework of abstract interpretation [103, 25, 28], a special case of which is predicate abstraction [47, 31, 50, 59]. We use concepts of abstract interpretation, such as for instance the theory of Galois connections [2]. In the conclusion we will discuss how the approach could be extended to predicate abstraction.

We will first introduce the algebra of attributes and attribute abstraction (Section 4.1). For the definition of signatures and algebras see also [115]. Then we describe attributed Petri nets and their analysis (Section 4.2). After this we show how to define and rewrite attributed graphs (Section 4.3). In Section 4.4 AGTSSs are approximated by attributed Petri nets and the necessary correctness results are proved. The results of this chapter have been published in [70].

4.1 Algebras and Abstraction

We first need a notion of a Σ -algebra. In the thesis we describe attributes of graph transformation systems and Petri nets as elements of a given Σ -algebra, where in the signature we describe possible data types as sorts of the algebra and possible operations as its function symbols. For example, if we need integer attributes, we consider a signature having a single sort *Integer* and the usual integer operations (such as plus and minus) as function symbols. The content of this section is based on the theory of signatures and algebras as described in [115] and on the abstract interpretation framework as described in [103, 28].

Definition 4.1.1 (Signature, Algebra) *A signature Σ is a pair $\langle \mathcal{S}, \mathcal{F} \rangle$ where \mathcal{S} is a set of sorts and \mathcal{F} is a set of function symbols equipped with a mapping $\sigma: \mathcal{F} \rightarrow \mathcal{S}^* \times \mathcal{S}$. Sorts will also be called types.*

A Σ -algebra \mathcal{A} consists of carrier sets $(\mathcal{A}_s)_{s \in \mathcal{S}}$ for each sort and a function $f^{\mathcal{A}} : \mathcal{A}_{s_1} \times \dots \times \mathcal{A}_{s_n} \rightarrow \mathcal{A}_s$ for every function symbol f with $\sigma(f) = (s_1 \dots s_n, s)$.

For a Boolean Σ -algebra we require that \mathcal{S} contains at least the sort *Bool* and that we have two subsets $T_{\mathcal{A}}, F_{\mathcal{A}} \subseteq \mathcal{A}_{\text{Bool}}$ denoting the truth values *true* and *false*.¹

For an algebra \mathcal{A} we denote by $\mathcal{A}_{\mathcal{S}}$ the set $\mathcal{A}_{\mathcal{S}} = \bigsqcup_{s \in \mathcal{S}} \mathcal{A}_s$, i.e., the union of all carrier sets (under the implicit assumption that they are all disjoint).

In our examples the sorts are *Bool*, *Int*, *Str*, *Unit* and tuples over the first three sorts. We use brackets to denote tuples, for instance $[Bool, Int, Str]$ is a valid sort in our signature Σ . We consider also constants (0-ary function symbols) for each type and the following set of function symbols F :

1. Logical operations (\vee, \wedge, \neg) on the sort *Bool*.
2. Integer operations ($+, -, *, /$) on the sort *Int* and comparison operations ($<, >, \leq, \geq, =$) from *Int* to *Bool*.
3. Concatenation (\cdot) on the sort *Str* and the length operator ($| \cdot |$) from *Str* to *Int*.
4. Projections π_i with $\sigma(\pi_i) = ([s_1, \dots, s_n], s_i)$ and constructions k_n with $\sigma(k_n) = (s_1 \dots s_n, [s_1, \dots, s_n])$ for disassembling and assembling tuples.

All operations in (1)–(3) are binary, apart from \neg (negation) and $| \cdot |$ (length) which are unary operations.

Over this signature we consider the following Σ -algebra \mathcal{C} : for *Bool*, *Int*, *Str* we take as carrier sets the booleans *true*, *false*, the integers and all strings over a fixed alphabet. Tuples are formed in the obvious way, for example $[true, 21, 'abc']$ is an element of $\mathcal{A}_{[Bool, Int, Str]}$. The carrier set of the sort *Unit* is $\{u\}$ consisting of a single element u , which is a dummy element denoting the absence of attributes.

All function symbols are interpreted in the natural way. We will sometimes abbreviate combinations of operations with projections and constructions. For instance $[+, -]$ is defined as $[+, -]([a, b]) = k_2(\pi_1(a) + \pi_1(b), \pi_2(a) - \pi_2(b))$, where $[a, b] \in \mathcal{A}_{[Int, Int]}$.

We will now define two special algebras needed in the following. The first algebra is a powerset algebra, where instead of the given carrier sets we use their powersets and extend the existing operations correspondingly. This kind of algebra is useful for the application of the framework of abstract interpretation to the notion of a Σ -algebra. Because in abstract interpretation only lattice-ordered carrier sets can be considered we use powersets with set inclusion as the order.

The second algebra - a term algebra - is a special case of a Σ -algebra, where only terms over a set of free variables are considered as carrier sets of the algebra. This kind of algebra is used for describing operations on attributes. In AGTSs the operations are described in the left-hand and right-hand sides of the rules as attributes of the corresponding hypergraphs and must therefore also be represented as elements of Σ -algebras (more specifically as elements of term algebras). Also the guards in both AGTSs and attributed Petri nets (and also the operations attached to the pre/post-sets of attributed Petri nets) will be terms of the corresponding term Σ -algebras.

Definition 4.1.2 (Powerset algebra) For a given Σ -algebra \mathcal{A} we will denote by $\mathcal{P}(\mathcal{A})$ its powerset algebra which is an algebra over the same signature. However the carrier

¹Note that here we do not restrict ourselves to two-valued logics. Three-valued logics with $T_{\mathcal{A}} = \{1, 1/2\}$ and $F_{\mathcal{A}} = \{0, 1/2\}$ is quite common in program analysis (see for instance [98]) and in this paper we will switch from an algebra to its powerset algebra, leading to a 4-valued logics. Furthermore the sets T, F need not form a partial of $\mathcal{A}_{\text{Bool}}$.

sets of $\mathcal{P}(\mathcal{A})$ are the powersets of the previous carrier sets, i.e., $\mathcal{P}(\mathcal{A})_s = \mathcal{P}(\mathcal{A}_s)$ and function symbols f with $\mathcal{F}(f) = (s_1 \dots s_n, s)$ are interpreted as follows:

$$f^{\mathcal{P}(\mathcal{A})}(A_1, \dots, A_n) = \{f^{\mathcal{A}}(a_1, \dots, a_n) \mid a_i \in A_i\},$$

where $A_i \in (\mathcal{P}(\mathcal{A}))_{s_i}$.

In the case of a Boolean algebra we set $T_{\mathcal{P}(\mathcal{A})} = \{A' \subseteq \mathcal{A}_{Bool} \mid A' \cap T_{\mathcal{A}} \neq \emptyset\}$ and similarly for $F_{\mathcal{P}(\mathcal{A})}$.

Note that in the case of our example algebra \mathcal{C} we have four truth values in $\mathcal{P}(\mathcal{C})$, where $T_{\mathcal{P}(\mathcal{C})} = \{\{true\}, \{true, false\}\}$, $F_{\mathcal{P}(\mathcal{C})} = \{\{false\}, \{true, false\}\}$. Going to powersets is a necessary step since the concretization of abstract values, which will be introduced later, provides us with an entire set of values, as opposed to a single value. However if we only start working with single values, i.e., one-element sets, we will get exactly the same results as in the original algebra.

Definition 4.1.3 (Term algebra) Let X be a set of free variables. A Σ -term algebra $T(\Sigma, X)$ over X has as carrier sets all terms of a specific sort. Furthermore function symbols are interpreted in a purely syntactical way.

Finally we need the notion of algebra homomorphism in order to define a correspondence relation between two Σ -algebras. This correspondence relation can for example be useful for describing matchings of attributed hypergraphs. Part of the match between two attributed hypergraphs is a match between their attributes, which can be represented by an algebra homomorphism.

Definition 4.1.4 (Algebra homomorphism) Let \mathcal{A}, \mathcal{B} be two Σ -algebras. An algebra homomorphism $h : \mathcal{A} \rightarrow \mathcal{B}$ is a family of maps $(h_s : \mathcal{A}_s \rightarrow \mathcal{B}_s)_{s \in \mathcal{S}}$ such that for each $f \in \mathcal{F}$ with $\sigma(f) = (s_1 \dots s_n, s_{n+1})$ we have

$$h_{s_{n+1}}(f^{\mathcal{A}}(a_1, \dots, a_n)) = f^{\mathcal{B}}(h_{s_1}(a_1), \dots, h_{s_n}(a_n)).$$

In our example algebra \mathcal{C} we consider infinite carrier sets for *Int*, *Str*. In order to analyse the systems thus obtained we need a mechanism of approximation. Here we work in the framework of abstract interpretation [29]. We start with the notion of a Galois connection.

Definition 4.1.5 (Galois connection on algebras) Let $\Sigma = \langle \mathcal{S}, \mathcal{F} \rangle$ be a signature and let \mathcal{A}, \mathcal{B} be two algebras over this signature, where each carrier set is lattice-ordered² via \sqsubseteq .

A family of functions $(\alpha_s : \mathcal{A}_s \rightarrow \mathcal{B}_s, \gamma_s : \mathcal{B}_s \rightarrow \mathcal{A}_s)_{s \in \mathcal{S}}$ is called Galois connection on algebras if they are monotone with respect to \sqsubseteq and if for all $s \in \mathcal{S}$:

1. $\forall a \in \mathcal{A}_s : a \sqsubseteq \gamma_s(\alpha_s(a))$
2. $\forall b \in \mathcal{B}_s : \alpha_s(\gamma_s(b)) \sqsubseteq b$

Finally we require that for each function symbol f with $\sigma(f) = (s_1 \dots s_n, s)$ the function $f^{\mathcal{B}}$ is a safe over-approximation of $f^{\mathcal{A}}$, i.e., for all a_1, \dots, a_n with $a_i \in \mathcal{A}_{s_i}$ it holds that:

$$\alpha_s(f^{\mathcal{A}}(a_1, \dots, a_n)) \sqsubseteq f^{\mathcal{B}}(\alpha_{s_1}(a_1), \dots, \alpha_{s_n}(a_n)).$$

²The partial order \sqsubseteq stands for the information ordering. Intuitively whenever $a \sqsubseteq b$, then a is considered to be more exact, i.e., a conveys more information about the system state.

Note that this condition says that α is an algebra homomorphism “up to” \sqsubseteq . Such mappings will also be called \sqsubseteq -homomorphisms.

Furthermore if \mathcal{A}, \mathcal{B} are Boolean we require that both use the same carrier set for *Bool*, that $\alpha_{\text{Bool}}, \gamma_{\text{Bool}}$ are identities, $T_{\mathcal{A}} = T_{\mathcal{B}}, F_{\mathcal{A}} = F_{\mathcal{B}}$ and that furthermore truth values respect the information ordering \sqsubseteq in the following sense: each value contained in the intersection of $T_{\mathcal{A}}$ and $F_{\mathcal{A}}$ is larger than any value only in $T_{\mathcal{A}}$ (or $F_{\mathcal{A}}$). And those in turn are larger than values not contained in any of the two sets.

Specifically we will use as \mathcal{A} the powerset algebra $\mathcal{P}(\mathcal{C})$ defined above and as \mathcal{B} an algebra of abstract values later denoted by \mathcal{C}^a .

In our examples we abstract attributes by an algebras having only finite carrier sets. This is needed in order to mechanize the verification procedures. We can not use the algebra \mathcal{C} for a Galois abstraction since it is not lattice-ordered. Hence we work with $\mathcal{P}(\mathcal{C})$ where the lattice-order is set inclusion.

In our tool we use at the moment following finite abstractions of integer attributes:

1. *Unit abstraction*: All numbers are considered as equal.
2. *Modulo abstraction*: Each number is considered modulo some pre-defined base.
3. *Sign abstraction*: We consider only the signs $(+, -, 0)$ of integer numbers.
4. *Interval abstraction*: The numbers between $-m$ and n are represented exactly. By $(+)$ and $(-)$ we denote the numbers larger and smaller than n and $-m$ respectively.

In order to obtain lattice-ordered sets of abstract values, each abstract algebra is also equipped with top and bottom elements (\top, \perp) . Unit abstraction can be seen as a case of modulo abstraction with a base one and sign abstraction can be seen as a case of interval abstraction with $n = m = 0$. Hence we consider in the following only modulo and interval abstractions. For these two abstractions we can define integer operations consistent with Definition 4.1.5. For example in the case of modulo abstraction, $i + j$ (where i and j are the abstract values) is $(i + j)(\text{mod } k)$. For the interval abstraction with $m = n = 5$ it holds that $2 + 3 = 5$, $3 + 3 = (+)$ and $(+) - (+) = \top$.

The Booleans will not be abstracted with respect to the powerset algebra. That is, we have the four boolean values $\emptyset, \{\text{true}\} = 1, \{\text{false}\} = 0, \{\text{true}, \text{false}\} = 1/2$. For example, for modulo abstraction the value of $2 < 3$ is $1/2$ and for interval abstraction (with $n \geq 3$) the value of $2 < 3$ is 1.

For sort *Str* we consider following abstractions.

1. *Unit abstraction*: All strings are considered as equal.
2. *Length abstraction*: The strings with length smaller than some n are represented exactly and all other strings are represented by $(+)$.

We also define the operations for these two types of abstractions. For example for the length abstraction the value of $a.b$ is equal to $a.b$ if $|a.b| \leq n$ and $(+)$ otherwise.

In the following an abstract algebra corresponding to our example algebra $\mathcal{P}(\mathcal{C})$ will be denoted by \mathcal{C}^a . Hypergraphs attributed with \mathcal{C}^a are called *attributed with abstract values*.

4.2 Attributed Petri Nets

In this section we give now a formal definitions of an attributed Petri net. We consider a fixed set of labels Λ and a function $ltype : \Lambda \rightarrow S$.

Let us remind here that by A^\oplus we denote a free commutative monoid over A with monoid operation \oplus , whose elements are also called *multisets*. A multiset $M \in A^\oplus$ can be written as a formal sum $M = \bigoplus_{a \in A} m_a \cdot a$. A function $f : A \rightarrow B$ can be extended to a function $f : A^\oplus \rightarrow B^\oplus$ on multisets.

Definition 4.2.1 (Attributed Petri net) Let \mathcal{A} be a Σ -algebra. An \mathcal{A} -attributed Petri net is a tuple

$$\mathcal{N} = (S, T, l, \bullet(), ()^\bullet, \text{guard}, m_0),$$

where S is a set of places, T is a set of transitions, $l : S \rightarrow \Lambda$ is a labelling function, $\bullet(), ()^\bullet : T \rightarrow (S \rightarrow (T(\Sigma, X)_S)^\oplus)$ are pre- and post-set functions, $\text{guard} : T \rightarrow T(\Sigma, X)_{\text{Bool}}$ is a guard function, and m_0 is the initial marking of the net.

A marking of an attributed Petri net is a function $m : S \rightarrow \mathcal{A}_S^\oplus$, where for each $s \in S$ each element of $m(s)$ is of sort $\text{ltype}(l(s))$.

The following well-typedness conditions should be satisfied:

1. Each element of the multisets $\bullet t(s)$ and $t^\bullet(s)$ is of sort $\text{ltype}(l(s))$.
2. The multiset $\bullet t(s)$ coincides with a set X' of variables where each element has multiplicity 1. Furthermore, the elements of $t^\bullet(s)$ are contained in $T(\Sigma, X')$ and $\text{guard}(t) \in T(\Sigma, X')_{\text{Bool}}$.
3. Each element of $m_0(s)$ is of sort $\text{ltype}(l(s))$.

Elements of $m(s)$ (which are elements of the carrier sets) are also called *tokens*. For a marking m define $|m| : S \rightarrow \mathbb{N}$ as $|m|(s) = |m(s)|$, i.e., each place is associated with the number of tokens it contains.

A transition t is *enabled* on the marking m if there exists a binding $h : T(\Sigma, X) \rightarrow \mathcal{A}_S$ such that $h(\text{guard}(t)) \in T_{\mathcal{A}}$ and for each place s it holds that $m(s) \geq h(\bullet t(s))$. An enabled transition with a given binding h can be fired and the marking m of the net will be transformed into m' in the following way:

$$m'(s) = m(s) \ominus h(\bullet t(s)) \oplus h(t^\bullet(s)).$$

We denote this by $m[t, h]m'$.

To analyse attributed Petri nets we adapted a forward coverability technique from Section 2.2 to attributed Petri graphs with finite domains of attributes. This technique is quite similar to standard coverability graphs; however, instead of tokens in places we count the occurrences of tokens with the same attribute in a place. We also introduce ω 's in order to describe unbounded numbers of tokens.

Fig. 4.1 represents a very simple attributed Petri net and the corresponding coverability graph. Here only a token "3" from place "A" can be used when firing the transition (because of the guard condition " $x > 2$ "), but this can be done infinitely many times because the token will be returned back into "A". In this way we obtain infinitely many tokens "4" in the place "B", which corresponds to the ω in the coverability graph.

A second possibility to analyse coverability in attributed Petri nets is an adaptation of the backward coverability algorithm from Section 2.4. Here the main difference to the usual backward coverability algorithm (Section 2.4) is that one can step backward for each transition in different ways, corresponding to combinations of attribute values.

Fig. 4.2 represents possible backward steps for a given marking of the attributed Petri net. We use here the interval abstraction of integers with an interval $[1, 2]$ and the value \top is a top element describing all integers. The table on the right shows what is generated in the places "B" and "C" after doing a backward step with the corresponding binding of x and y . The binding $x = 1$ is not allowed because it violates the guard condition.

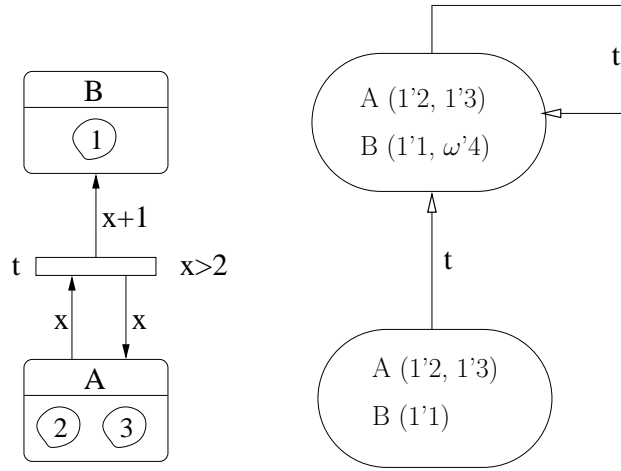


Figure 4.1: Attributed Petri net and the corresponding coverability graph

Let us consider for example a binding $x = 2$ and $y = 1$. After firing of transition we obtain 2 tokens $2 + 1 = \top$ in "B" and one token $1 + 1 = 2$ in "C". This means in the step backward one removes the existing tokens \top from "B" and "2" from "C". After this the tokens 2 and 1 can be added to the places "A" and "D" correspondingly. In the table in Fig 4.2 the possible bindings of x and y and the corresponding obtained markings in places "B" and "C" are presented.

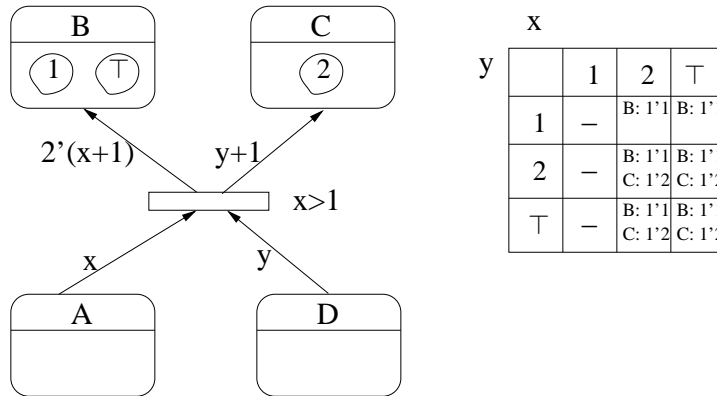


Figure 4.2: Backward step in the attributed Petri net

The exact description of the algorithms is out of the scope of this paper. For the abstraction refinement procedure (see Section 5.2) we often need a trace (counterexample) to a given marking, which can be obtained in the case of attributed Petri nets in the same way it was done for standard Petri nets.

Both forward and backward coverability algorithms for attributed Petri nets are implemented in AUGUR 2.

4.3 Attributed Graph Transformation Systems

We consider a fixed typing function $ltype : \Lambda \rightarrow \mathcal{S}$ which associates a sort to each label. Note that the theory can be easily extended to associating several (named) attributes

to each label and this is also the way it is handled in our implementation. In order to obtain a simpler presentation we will here only use one attribute per label.

Definition 4.3.1 (attributed hypergraph) *A (labelled) hypergraph attributed over a Σ -algebra \mathcal{A} is a tuple $G = (V_G, E_G, c_G, l_G, attr_G)$, where (V_G, E_G, c_G, l_G) is a labelled hypergraph and $attr_G : E_G \rightarrow \mathcal{A}_S$ is a function such that for each edge $e \in E_G$ it holds that $attr_G(e) \in \mathcal{A}_{l_{type}(l_G(e))}$.*

We consider nodes of a hypergraph as unlabelled (and without attributes). Attributes can be added to nodes by providing nodes with unary hyperedges which keep the attribute for that node³.

For now we consider the following two types of attributed hypergraphs, where \mathcal{A} is called algebra of attributes:

1. *Value-attributed hypergraphs* with a concrete algebra \mathcal{A} . In our examples it is often $\mathcal{A} = \mathcal{C}$ or $\mathcal{A} = \mathcal{P}(\mathcal{C})$ (whenever $\mathcal{A} = \mathcal{P}(\mathcal{C})$, the graph will also be called powerset-attributed). In this case we require that \mathcal{A} is a Boolean algebra.
2. *Term-attributed hypergraphs* with $\mathcal{A} = T(\Sigma, X)$.

Now we recall the definition of a hypergraph morphisms from Section 3.1 and extend it to the definition of a morphisms between attributed hypergraphs.

Definition 4.3.2 (hypergraph morphisms) *Let G_1 and G_2 be two hypergraphs. A (hypergraph) morphism $\varphi : G_1 \rightarrow G_2$ consists of two total functions $\varphi_V : V_{G_1} \rightarrow V_{G_2}$ and $\varphi_E : E_{G_1} \rightarrow E_{G_2}$ such that for every $e \in E_{G_1}$ it holds that $l_{G_1}(e) = l_{G_2}(\varphi_E(e))$ and $\varphi_V^*(c_{G_1}(e)) = c_{G_2}(\varphi_E(e))$. A morphism is called edge-bijective whenever it is bijective on edges.*

Definition 4.3.3 (morphisms of attributed hypergraphs) *Let G and G' be two attributed hypergraphs (where G is attributed over \mathcal{A} and G' over \mathcal{B}). An attributed hypergraph morphism $\varphi = (\varphi_V, \varphi_E, h) : G_1 \rightarrow G_2$ consists of a hypergraph morphism (φ_V, φ_E) and an algebra homomorphism $h : \mathcal{A} \rightarrow \mathcal{B}$ such that*

$$\forall e \in E_{G_1} : attr_{G_2}(\varphi_E(e)) = h(attr_{G_1}(e)).$$

Attributed hypergraphs can be transformed using rewriting rules which we define in the following way. We use the same restrictions on rules as in [6] since this greatly simplifies the verification procedure.

Definition 4.3.4 (attributed rewriting rule) *We fix a signature Σ and a set X of variables. An attributed rewriting rule r is a quadruple (L, R, α, g) , where L and R are term-attributed hypergraphs, called left-hand side and right-hand side respectively, and $\alpha : V_L \rightarrow V_R$ is an injective mapping, indicating how nodes are preserved.*

We demand that each of the term attributes of L is a single variable of X (such that each variable appears only once). The set of all variables in the left-hand side is denoted by X' . The right-hand side R may be attributed with arbitrary terms from $T(\Sigma, X')$. Each rule r is associated with a guard expression $g(r) \in T(\Sigma, X')_{Bool}$.

We demand also that there are no isolated nodes in the left-hand side L and no isolated nodes in $V_R - \alpha(V_L)$. Additionally E_L must not be empty. Furthermore there can not be two edges with the same label on the left-hand side of a rule.

³Note that here we choose a different representation of attributed graphs than in [40] where the focus is on viewing attributed graphs in the framework of adhesive HLR categories and where graphs include specific data nodes. One of our main concerns is to fully separate the graph structure and the attributes for verification purposes.

If an instance of the left-hand side is found in the current state of the system, then this rule can be applied and the instance of the left-hand side of the rule will be replaced by its right-hand side. We are now ready to define the notion of attributed graph transformation systems.

Definition 4.3.5 (attributed graph transformation system (AGTS)) An attributed graph transformation system (AGTS) $\mathcal{G} = (\mathcal{R}, G_0)$ is a finite set of attributed rewriting rules \mathcal{R} together with a value-attributed start hypergraph G_0 (also called initial graph).

Definition 4.3.6 (rule application) A match of a rewriting rule $r = (L, R, \alpha, g)$ in a graph G (which is value-attributed over an algebra \mathcal{A}) is a morphism $\psi = (\varphi, h) : L \rightarrow G$ which is injective on edges. We can apply r to a match in G obtaining a new graph H , written $G \xRightarrow{r} H$, whenever the guard expression is satisfied, i.e., $h(g) \in T_{\mathcal{A}}$. The target graph H is defined as follows

$$V_H = V_G \uplus (V_R - \alpha(V_L)) \quad E_H = (E_G - \varphi(E_L)) \uplus E_R$$

and, defining $\overline{\varphi} : V_R \rightarrow V_H$ by $\overline{\varphi}(\alpha(v)) = \varphi(v)$ if $v \in V_L$ and $\overline{\varphi}(v) = v$ otherwise, the source, target and labelling functions are given by

$$\begin{aligned} e \in E_G - \varphi(E_L) &\Rightarrow c_H(e) = c_G(e), \quad l_H(e) = l_G(e) \\ e \in E_R &\Rightarrow c_H(e) = \overline{\varphi}(c_R(e)), \quad l_H(e) = l_R(e) \end{aligned}$$

Additionally we define

$$\begin{aligned} e \in E_G - \varphi(E_L) &\Rightarrow attr_H(e) = attr_G(e) \\ e \in E_R &\Rightarrow attr_H(e) = h(attr_R(e)) \end{aligned}$$

That is, a left-hand side is found and replaced by the corresponding right-hand side. We use restricted version of the DPO (double-pushout) approach where we only allow discrete interfaces. Merging as well as deletion of nodes is forbidden. Edges, however, can be deleted. The new attributes in the right-hand side are obtained by using h , the binding of the set of free variables X' of the left-hand side. The value $h(t)$ for a term $t = attr_R(e)$ is also called an *interpretation* of t in the algebra \mathcal{A} .

Note that in the case of a powerset algebra, some elimination of over-approximation could be useful. For instance, we could remove values that do not satisfy the guard expression. In order to be able to represent the theory in a compact way we choose not to follow this path at the moment.

Fig. 4.3 shows a very simple example of an AGTS and a possible rewriting sequence for this example. For example, the edge with label A has two attributes $a1$ and $a2$. The first has type Int and the second is of type $[Int, Str]$. The rewriting sequence consists only of two steps because in the last hypergraph the expression $y > 0$ is evaluated to false and therefore the rule can not be applied another time. Note that in this example we use a more general approach, where the typing function $ltype$ associates several sorts to each label, for example, the edge with label A has two attributes $a1$ and $a2$. As mentioned before the theory can be easily extended to this case and actually in our implementation things are handled in this way. This extension is simple because according to Definition 4.3.4 each of the term attributes of left-hand sides of rewriting rules is a single variable (such that each variable appears only once).

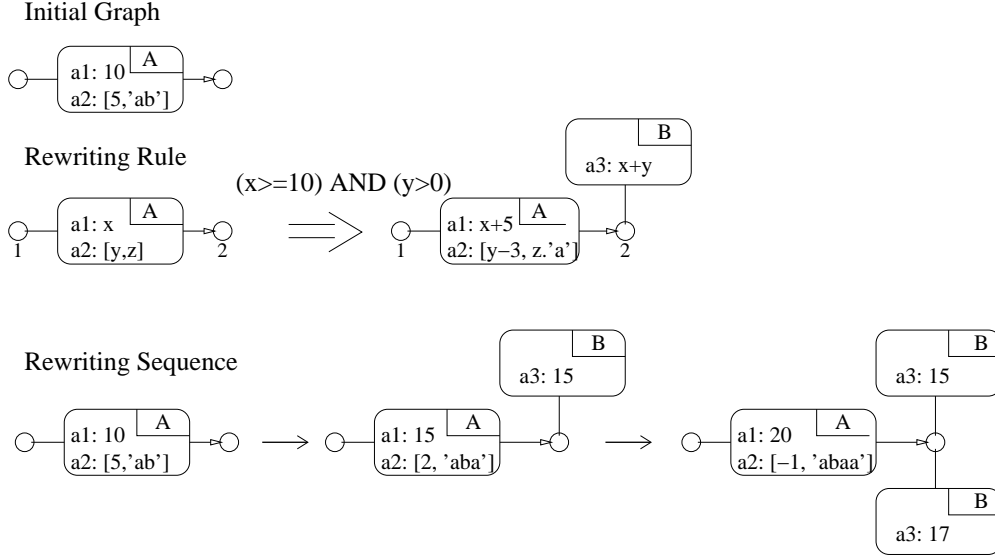


Figure 4.3: First example of an attributed graph transformation system

4.4 Verification of Attributed Graph Transformation Systems

Since GTSSs are in general Turing-powerful, over-approximation techniques are needed for their analysis. In our case we abstract AGTSs by coloured Petri nets, which are a conceptually simpler formalism, for which several verification techniques have already been developed. In Section 3.2 the approximated unfolding technique for GTSSs was presented. Compared to a standard unfolding technique additional folding steps are used which over-approximate and guarantee a finite approximation. The resulting over-approximation is a so-called Petri graph which is a Petri net having a hypergraph structure additionally. The hyperedges are at the same time the places of the net. Our idea here is to construct an *attributed* Petri graph which over-approximates an AGTS: an attributed Petri graph consists of an attributed (or coloured) Petri net and a hypergraph structure over it.

Our notation is oriented on coloured Petri nets developed by Jensen [58]. We also adapt two existing analysis techniques for Petri nets (coverability graph and backward reachability algorithms, Sections 2.2 and 2.4) to the case of attributed Petri nets. These techniques can be used only in the case of finite carrier sets, which arise from the abstraction as described in Section 4.1.

In order to demonstrate our verification technique, we use a simple example AGTS shown in Fig. 4.4. There are two integer attributes in this example: n for edges labelled B and c for edges labelled C . Attribute n is increased by one on the newly created edge B . Attribute c is multiplied with the corresponding attribute n when C crosses B . The edges A and Error have empty sets of attributes. The property we want to verify is that no Error edge will ever be created.

We give now a formal definitions of an attributed Petri graph. We consider a fixed set of labels Λ and a function $ltype : \Lambda \rightarrow S$.

We consider a Petri graph as consisting of an attributed Petri net and a non-attributed hypergraph structure over it.

Definition 4.4.1 (attributed Petri graph) Let $\mathcal{G} = (\mathcal{R}, G_0)$ be an AGTS. An \mathcal{A} -

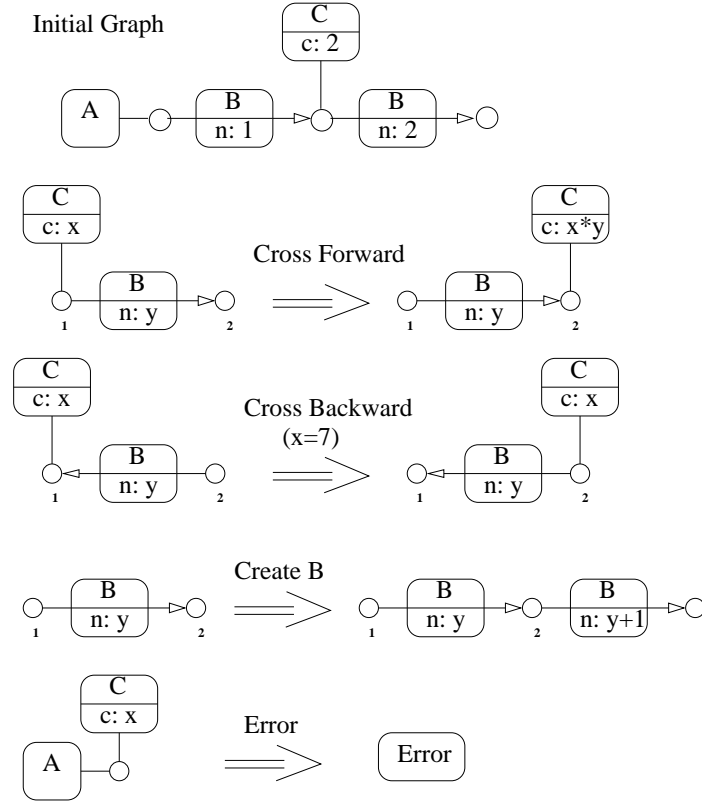


Figure 4.4: Second example of an attributed graph transformation system

attributed Petri graph (over \mathcal{R}) is a tuple $P = (G, \mathcal{N}, p_N, \mu)$, where G is a (non-attributed) hypergraph, \mathcal{N} is an \mathcal{A} -attributed Petri net where the places are the edges of G , p_N associate to each transition t a rule $p_N(t) = (L, R, \alpha, g) \in \mathcal{R}$ such that $\text{guard}_P(t) = g$ and μ associates to each transition t from \mathcal{N} with $p_N(t)$ as above a (non-attributed) hypergraph morphism $\mu(t) : L \cup R \rightarrow G$ ⁴ such that $\bullet t(s) = \bigoplus_{\mu(t)(e)=s, e \in E_L} \text{attr}_L(e)$ and $t^\bullet(s) = \bigoplus_{\mu(t)(e)=s, e \in E_R} \text{attr}_R(e)$.

An attributed Petri graph for the GTS \mathcal{G} is a pair (P, ι) , where $P = (G, \mathcal{N}, p_N, \mu)$ is an attributed Petri graph over \mathcal{R} and $\iota : G_0 \rightarrow G$ is an attributed graph morphism such that $m_0(e') = \bigoplus_{e \in i^{-1}(e')} \text{attr}_{G_0}(e)$ for each edge $e' \in E_{G_0}$.

Note that the edges of the graph are at the same time the places of the net and that the transitions are labelled with rules of the AGTS. The meaning of morphisms ι and $\mu(t)$ is the same as in the non-attributed Petri graph: ι describes a subgraph corresponding to the initial graph of AGTS and $\mu(t)$ describes, for each transition t , the relation between the left/right-hand sides of an attributed rewriting rule $p_N(t)$ and the corresponding subgraphs of AGTS. We consider the name of the rule $p_N(t)$ to be a label for transition t .

For each marking m of an attributed Petri graph we define an attributed graph $\text{graph}(m)$ in the following way: First we take the subgraph of G having only edges E' such that $m(e') \neq \emptyset$ for each $e' \in E'$ and the nodes V' connected with E' . Let $m(e') = \sum_{i=1}^k a_i$ be the marking of an edge $e' \in E'$ (some elements in the sum can be equal). Instead of each edge $e' \in E'$ we insert k edges e_1, \dots, e_k such that $l_G(e_i) = l_G(e)$, $c_G(e_i) = c_G(e)$ and $\text{attr}_G(e_i) = a_i$.

We describe now how to obtain an attributed Petri graph from the given AGTS. First we unfold the underlying GTS in the approximative way as it is described in Section 3 without taking attributes into consideration. This is done by applying the usual unfolding steps as well as folding steps that merge left-hand sides which are causally dependent. Since the system under consideration is in general infinite-state, these folding steps are necessary in order to keep the approximation finite. Since the approximated unfolding procedure supplies us with morphisms ι and $\mu(t)$ as described in Definition 3.2.1 there is a unique way of adding attributes to the Petri graph after the approximated unfolding (i.e., attributes do not affect the unfolding procedure itself in any way).

We add attributes to the Petri graph as specified by Definition 4.4.1. This is depicted in Figures 4.5 and 4.6. Here we first use all attributes from the algebra \mathcal{A} of concrete values from the value-attributed initial hypergraph of AGTS in the corresponding places of the Petri graph as tokens (Fig 4.5). The correspondence here is given by the morphism ι . If $\iota : G_0 \rightarrow G$, $e \in E_{G_0}$ and $\text{attr}_{G_0}(e) = a$ then we put a token a into $\iota(a)$. Although theoretically we consider only the case of a single attribute type for each label, in the example we show the more general case of multiple attributes in order to underline the fact, that all attribute values from a single edge are composed into a single token.

Fig. 4.6 shows how the values from a term algebra of attributes from the rules of AGTS are attached to the arcs of the constructed Petri graph. The attributes here are described as values of the multisets obtained from the pre- and post-sets of the transitions and are depicted on the arcs connecting places and transitions. Formally, for each transition t with $p_N(t) = (L, R, \alpha, g)$ and $\mu(t) : L \cup R \rightarrow G$ we attach preset and postset attributes (according to Definition 4.4.1):

$$\bullet t(s) = \bigoplus_{\mu(t)(e)=s, e \in E_L} \text{attr}_L(e), \quad t^\bullet(s) = \bigoplus_{\mu(t)(e)=s, e \in E_R} \text{attr}_R(e)$$

⁴We use the same function ltype for \mathcal{G} and \mathcal{N} which gives us the correspondence of attributes in $L \cup R$ to the attributes in $\mu(t)(L \cup R)$

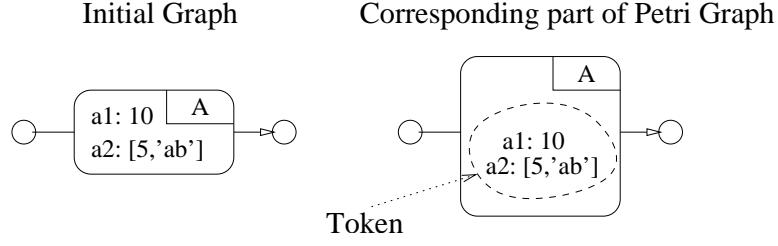


Figure 4.5: Setting initial marking to Petri graph (based on example in Fig. 4.3)

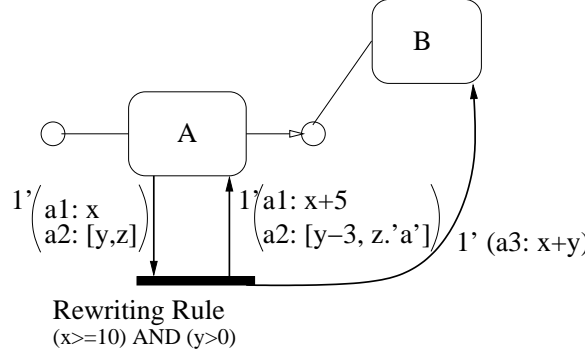


Figure 4.6: Example of setting attributes to the Petri graph (based on example in Fig. 4.3)

In the case of multiple attributes (as also in the case of the initial marking) all attributes of a single edge are combined.

Finally, the guards of the transitions of a Petri graph are attached, based on the guards of the corresponding rules of the AGTS. For transition t in the unfolding with $p_N(t) = r$ we use the guard of r as guard of the transition.

The attributes are added to the Petri graph at the end of the unfolding procedure (i.e., attributes do not affect the unfolding procedure itself). This means that the confluence and termination results from Section 3.2 obtained for non-attributed Petri graphs can be also used in the case of attributed Petri graphs, but we still need to prove the correctness result.

Before we describe the correctness statement we define a notion of abstract runs for attributed GTs and attributed Petri graphs and also the notion of the correspondence between abstract runs.

Definition 4.4.2 (Abstract run of an AGTS) *An abstract run of an AGTS (\mathcal{R}, G_0) is a sequence of attributed hypergraphs $\mathcal{J} = (J_0 \Rightarrow_{r_1} J_1 \Rightarrow_{r_2} \dots \Rightarrow_{r_n} J_n)$, where r_i is a rule name, together with (attributed) morphisms $\varphi_i : L_{i+1} \rightarrow J_i$ for each $i = 1, \dots, n-1$, where L_i is the left-hand side of rule $r_i \in \mathcal{R}$.*

Note that we do not demand that J_i can be derived from J_{i-1} by applying rule r_i at match φ_i (hence the name abstract). If an abstract run is derivable it will be called a real run.

The j -th prefix of \mathcal{J} is the run $pr_j(\mathcal{J}) = (J_0 \Rightarrow_{r_1} J_1 \Rightarrow_{r_2} \dots \Rightarrow_{r_j} J_j)$ together with the morphisms φ_i .

Let $\mathcal{J}' = (J'_0 \Rightarrow_{r_1} J'_1 \Rightarrow_{r_2} \dots \Rightarrow_{r_n} J'_n)$ be another abstract run with morphisms $\varphi'_i : L_{i+1} \rightarrow J'_i$ for each $i = 1, \dots, n-1$. We say that \mathcal{J}' weakly corresponds to \mathcal{J} (in symbols $\mathcal{J}' \ll \mathcal{J}$) if for each $i = 1, \dots, n-1$ there exist edge-bijective (attributed) morphisms $\xi_i : J'_i \rightarrow J_i$ for $i = 0, \dots, n$. If furthermore the following diagram commutes

we say that \mathcal{J}' corresponds to \mathcal{J} and write $\mathcal{J}' \lll \mathcal{J}$.

$$L_{i+1} \xrightarrow{\varphi'_i} J'_i \xrightarrow{\xi_i} J_i$$

$$\quad \quad \quad \searrow \varphi_i$$

In both cases, if the attributed morphisms are not equipped with algebra homomorphisms, but just \sqsubseteq -homomorphisms (as defined in Definition 4.1.5) we talk about (weak) \sqsubseteq -correspondence and write \ll_{\sqsubseteq} and \lll_{\sqsubseteq} .

If it is not evident from a context we will write explicitly which algebra is used.

The definition above is closely related to the definitions of abstract run and correspondence between runs for (non-attributed) GTs (Definition 3.2.15). The main difference is that the morphisms φ_i here are attributed, i.e., they are additionally equipped with an algebra homomorphism h . We also consider here the weaker forms of morphisms by using \sqsubseteq -homomorphisms instead of algebra morphisms. The correspondence between two abstract attributed runs obtained in this case is called \sqsubseteq -correspondence and is used later in order to describe the relation between abstract runs with concrete and abstract attributes. The usage of \sqsubseteq -correspondence is essential in this case because applying the operation f_A to concrete values and the operation f_B to abstract values only guarantees us the \sqsubseteq -relation between $\alpha_s(f_A(a_1, \dots, a_n))$ and $f_B(\alpha_{s_1}(a_1), \dots, \alpha_{s_n}(a_n))$ (see Section 4.1).

Hypergraphs of an abstract run can be attributed either with the algebra of concrete values or with the algebra of abstract values or even with the term algebra. If it is not evident from a context we will write explicitly which algebra is used.

Petri graphs can, as mentioned above, be seen as symbolic representations of graph transition systems and also as representations of sets of abstract runs. The following definition describes abstract runs of an attributed Petri graph and is an extension of Definition 3.2.16 for (non-attributed) GTs.

Definition 4.4.3 (Abstract runs of an attributed Petri graph)

Let (P, ι) with $P = (G, N, p_N, \mu)$ be an attributed Petri graph for an AGTS (\mathcal{R}, G_0) . Furthermore let $m_0[t_1, h_1] \dots [t_n, h_n]m_n$ be a firing sequence of the net N and let $r_i = p_N(t_i)$ be the rules corresponding to the transitions. We consider (non-attributed) morphisms $\nu_{i+1} = e_{m_i, \mu(t_{i+1})|L_{i+1}} : L_{i+1} \rightarrow \text{graph}(m_i)$, where L_{i+1} is the left-hand side of rule r_{i+1} and extend them in the canonical way to attributed morphisms by adding bindings. It is easy to see that the sequence $\text{graph}(m_0) \Rightarrow_{r_1} \text{graph}(m_1) \Rightarrow_{r_2} \dots \Rightarrow_{r_n} \text{graph}(m_n)$ together with the morphisms $\varphi_i = (\nu_i, h_i)$ is an abstract run.

Each real run $\mathcal{J}_R = (G_0 \Rightarrow_{r_1} G_1 \Rightarrow_{r_2} \dots \Rightarrow_{r_n} G_n)$ of the AGTS (\mathcal{R}, G_0) can be considered as an abstract run where the $\varphi_i : L_{i+1} \rightarrow G_i$ represent the matches of the left-hand sides of the rules r_i .

In order to prove the correctness property we should show the following statement:

Proposition 4.4.4 (Correctness) *Let \mathcal{G} be an AGTS and let P be an attributed Petri graph approximating \mathcal{G} . Then, for every real run \mathcal{J}_R of \mathcal{G} there exists an abstract run \mathcal{J}_A of P , such that $\mathcal{J}_R \lll \mathcal{J}_A$.*

Proof: This holds since the firing of the Petri graph mimics the transformation in the graph transformation rules. The construction starts with the initial graph and every coverable left-hand side is unfolded at some point.

The attributes in the initial marking of P are exactly the same as in the start graph of \mathcal{G} . Furthermore by Definition 4.4.1 the guards and the attributes in the pre- and post-set of each transition in P coincide with the guards and attributes in the left- and right-hand sides in \mathcal{G} .

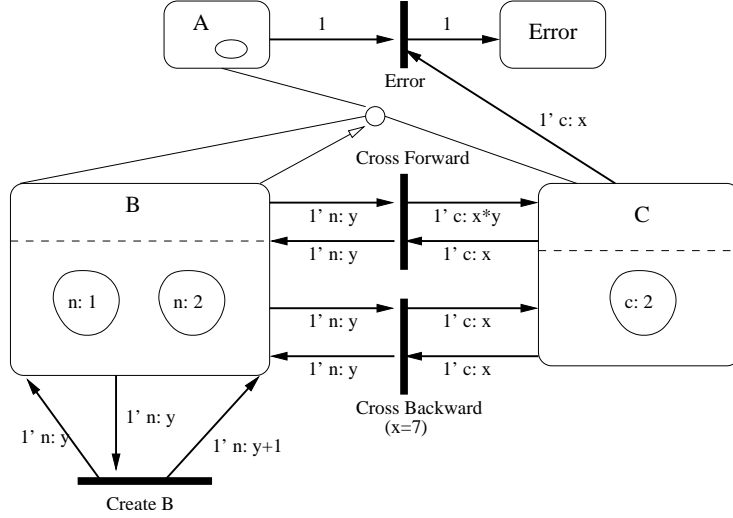


Figure 4.7: Petri graph approximating the GTS (first approximation).

This means that each application of a rule in \mathcal{G} can be simulated in P by firing the corresponding transition with exactly the same attributes.

Due to the merging of places in P new combinations of attributes can appear which can lead to spurious runs in P which have no correspondence in \mathcal{G} , but we can at least simulate all runs of \mathcal{G} in P . In this way each reachable hypergraph G corresponds a reachable marking m with the same attributes and there exists an edge-bijective attributed hypergraph morphism between G and $graph(m)$. \square

Fig. 4.7 depicts the coarsest over-approximation for the AGTS in Fig. 4.4.

The obtained Petri graphs are basically coloured Petri nets [58] and can be analyzed with techniques developed for such nets. First we will spell out what kind of properties can be verified using our over-approximation technique.

From Proposition 4.4.4 it follows that if a marking m is not reachable then there exists no reachable graph G in AGTS such that G can be mapped to $graph(m)$ via an attributed hypergraph morphism. We can also make the corresponding conclusion about subgraphs if we consider coverability instead of reachability. The problem is that since we have infinite domains of attributes we can analyse neither reachability nor coverability. Hence attribute abstraction is needed. We show here that if the attributes are correctly approximated (by using Galois connections as described above) then the non-reachability (non-coverability) of a marking in the Petri net with abstracted attributes means non-reachability (non-coverability) of a marking in the Petri net without abstraction of attributes.

In the following we assume that all AGTSs and Petri graphs are attributed over an algebra \mathcal{A} (in our examples: $\mathcal{P}(\mathcal{C})$). We use an abstract algebra \mathcal{B} (in our examples: \mathcal{C}^a) related to \mathcal{A} via a Galois connection (α_s, γ_s) . This also means that both algebras must be lattice-ordered and satisfy the requirements of Definition 4.1.5.

If we take a Petri graph attributed over \mathcal{A} this can be easily seen as a Petri graph attributed over \mathcal{B} by converting the initial marking via α_s . More precisely, an abstract marking $m^a : S \rightarrow (\mathcal{B}_S)^\oplus$ is obtained from a concrete marking $m : S \rightarrow (\mathcal{P}(\mathcal{C}))_S^\oplus$ by replacing every element a of $m(s)$ by $\alpha_{ltype(l(s))}(a)$. Note that in a Petri graph attributed over \mathcal{B} one also uses the abstract operations of \mathcal{B} (i.e., the abstract interpretation of the function symbols from \mathcal{F}).

Let us denote by P^a an (abstract) Petri graph obtained from a (concrete) Petri graph P as described above.

The following proposition shows how the abstract Petri graph P^a can be used in order to analyse P . We write $\hat{m}_1 \sqsubseteq \hat{m}_2$ for markings of P^a whenever they have the same number of tokens in each place and there is a bijective assignment $\eta_{\hat{m}_1, \hat{m}_2}$ (with $\eta_{\hat{m}_1, \hat{m}_2}(s) : \hat{m}_1(s) \rightarrow {}^\oplus \hat{m}_2(s)$ for each $s \in S$) of the tokens of \hat{m}_1 to the tokens in \hat{m}_2 such that each token v in the first marking has an abstract value that is smaller or equal (with respect to the lattice-order \sqsubseteq) than the associated token in \hat{m}_2 , i.e., $v \sqsubseteq \eta_{\hat{m}_1, \hat{m}_2}(s)(v)$.

Proposition 4.4.5 (Abstraction of attributes in Petri graph) *For the attributed Petri graphs P and P^a it holds that:*

1. *From the reachability (coverability) of m in P follows the reachability (coverability) of a marking \hat{m} with $\hat{m} \sqsupseteq m^a$ in P^a .*
2. *If $m_1[t, h]m_2$ in P and $\hat{m}_1 \sqsupseteq m_1^a$, then there exists \hat{m}_2 such that $\hat{m}_1[t, \hat{h}]\hat{m}_2$ in P^a and $\hat{m}_2 \sqsupseteq m_2^a$.*

Proof: The proposition holds since we designed the Galois connection in such a way that the functions of \mathcal{B} safely over-approximate the functions of \mathcal{A} .

Due to the information ordering on the Boolean values, whenever a guard term g is evaluated to true in \mathcal{A} , the same holds for its evaluation in \mathcal{B} (but not necessarily vice versa). In more detail: $g(a_1, \dots, a_n) = \alpha_{Bool}(g(a_1, \dots, a_n)) \sqsubseteq g(\alpha(a_1), \dots, \alpha(a_n))$ and therefore $g(a_1, \dots, a_n) \in T_{\mathcal{A}} \Rightarrow g(\alpha(a_1), \dots, \alpha(a_n)) \in T_{\mathcal{B}}$.

This means that every transition t that can be fired at m_1 can also be fired at the marking $\hat{m}_1 \sqsupseteq m_1^a$, resulting in markings m_2, \hat{m}_2 . We assume that the inequation $\hat{m}_1 \sqsupseteq m_1^a$ is witnessed by the bijection $\eta_{m_1^a, \hat{m}_1}$ on multisets.

Furthermore it can be shown that $\hat{m}_2 \sqsupseteq m_2^a$, as follows: For the marking m_2 it holds that

$$m_2(s) = m_1(s) \ominus h(\bullet t(s)) \oplus h(t^\bullet(s)),$$

where h is the corresponding binding.

We define an abstract binding $h^a(x) = \alpha(h(x))$ for each variable $x \in X$, which can be extended to $h^a : T(\Sigma, X) \rightarrow \mathcal{B}_{\mathcal{S}}$. Using the fact that $\bullet t(s)$ consists only of single variables from X we obtain $\alpha(h(\bullet t(s))) = h^a(\bullet t(s))$ and in addition $\alpha(h(t^\bullet(s))) \sqsubseteq h^a(t^\bullet(s))$ since α safely over-approximates the functions of \mathcal{A} . Hence we obtain:

$$\begin{aligned} m_2^a(s) &= \alpha(m_2(s)) = \alpha(m_1(s) \ominus h(\bullet t(s)) \oplus h(t^\bullet(s))) \sqsubseteq \\ & m_1^a(s) \ominus h^a(\bullet t(s)) \oplus h^a(t^\bullet(s)) \end{aligned}$$

Now we define a binding \hat{h} corresponding to the marking \hat{m}_1 by setting $\hat{h}(x) = \eta_{m_1^a, \hat{m}_1}(s)(h^a(x))$ for every variable x contained in $\bullet t(s)$. This means that t can be fired at \hat{m}_1 with binding \hat{h} resulting in \hat{m}_2 .

It follows that $\hat{h}(\bullet t(s)) = \eta_{m_1^a, \hat{m}_1}(s)(h^a(\bullet t(s)))$ and $\hat{h}(t^\bullet(s)) \sqsupseteq \eta_{m_1^a, \hat{m}_1}(s)(h^a(t^\bullet(s)))$ from the definition of \sqsubseteq .

Using $m_1^a \sqsubseteq \hat{m}_1$ and again the fact that $\bullet t(s)$ consists only of single variables from X we get:

$$m_2^a(s) \sqsubseteq \hat{m}_1(s) \ominus \hat{h}(\bullet t(s)) \oplus \hat{h}(t^\bullet(s)) = \hat{m}_2(s).$$

□

Let \mathcal{G} be an AGTS, let P be an attributed Petri graph approximating \mathcal{G} and let P^a be the abstract Petri graph derived from P . From the correctness property we obtain that for every real run \mathcal{J}_R of \mathcal{G} there exists an abstract run \mathcal{J}_A of P , such that $\mathcal{J}_R \lll \mathcal{J}_A$.

And furthermore for every abstract run \mathcal{J}_A of P there exists an abstract run $\hat{\mathcal{J}}_A$ of P^a such that $\mathcal{J}_A \ll \hat{\mathcal{J}}_A$. This is a direct consequence of Propositions 4.4.4 and 4.4.5. Since correspondence is transitive this means that every real run \mathcal{J}_R of \mathcal{G} can be associated with an abstract run $\hat{\mathcal{J}}_A$ of P^a such that $\mathcal{J}_R \ll \hat{\mathcal{J}}_A$.

In the verification of AGTSs the propositions above can, for example, be used in the following way: If one has proved that an error-rule having a "bad" attributed hypergraph in the left-hand side and a single (usually non-attributed) error-edge in the right-hand side cannot be applied in the evaluation of the abstract attributed Petri graph P^a (error-edge is not coverable), then the rule can also not be applied in the evaluation of the concrete attributed Petri graph P . In this way if a property (the absence of a "bad" graph in the evaluation) holds in P^a , then it also holds in P . The meaning of using a special error rule for the verification of (attributed) GTSs will be also described in the next section.

If the property does not hold, then we obtain a counterexample. In the next section we show how to deal with the obtained counterexample and how to eliminate it.

Chapter 5

Counterexample-Guided Abstraction Refinement

In the last years verification techniques based on counterexample-guided abstraction refinement [24] have been very successful. The technique has been used successfully in several tools such as SLAM [14], BLAST [50] or MAGIC [23]. The idea behind this approach is to start with a coarse initial abstraction or over-approximation of a system and to check whether a certain property can be verified using this abstraction. If it cannot be verified, one obtains a run in the approximation that violates this property, also called a counterexample.

Now either this counterexample is real or it is spurious, i.e., it has been introduced by the approximation. In the latter case the approximation is refined in such a way that the counterexample disappears. This process is repeated; however, in the case of infinite-state systems there is in general no guarantee that it will terminate, since the properties to be verified are usually undecidable.

In this chapter we describe the main theoretical result of this thesis, namely, an application of the counterexample-based abstraction refinement technique to non-attributed GTs (Section 5.1) and to attributed GTs (Section 5.2).

5.1 Abstraction Refinement of Graph Transformation Systems

In this section we describe the counterexample-guided abstraction refinement technique for GTs. Based on the notion of abstract run we define here a spurious abstract run which is in our case obtained from the over-approximation of GTs by Petri graphs. An abstract run \mathcal{J} for which there does not exist a real run corresponding to \mathcal{J} is called *spurious*. If, at the same time, it violates the property we attempt to verify, it is called a *counterexample* or *error trace*.

In Section 3.2 we have described another possibility to eliminate the spurious counterexample, namely, the using of the depth-based approach (see also [12]), where one constructs an over-approximation exact up to a predefined depth of the unfolding. The counterexample-guided abstraction refinement approach developed in the thesis compared to the depth-based approach usually results in smaller approximations and faster verification. The two approaches are compared in this thesis in Section 7.1 and Section 7.2 based on two examples: "Public Private Servers" and "Firewall".

Usually a counterexample is obtained by checking coverability of some final marking

(corresponding to some error sub-graph) in a Petri graph (see Section 2.2). It means we do not consider the hypergraph structure of the Petri graph and work only with its Petri net component. This can lead to problems because instead of the error sub-graph we consider only the set of its edges (which are places in the Petri net component). It is easy to avoid this problem by adding a special error rule which has the error sub-graph as a left-hand side and a single error edge with arity 0 as a right-hand side. In this case we always search for counterexamples leading to the error edge.

In order to eliminate spurious runs, we will show that they are always caused by the fact that certain nodes were merged. This is similar to the concept of summary nodes in shape analysis [98]. We will identify these nodes and show how to avoid their being merged in the next iteration, thereby avoiding this particular spurious run and all other abstract runs corresponding to it in a sense made precise later. We describe here how to construct a more exact over-approximation by separating merged nodes such that these spurious runs disappear. This procedure can be performed repeatedly for any number of spurious runs.

Merging nodes is harmful since it might produce new left-hand sides, thereby leading to additional rewriting steps. On the other hand, merging of edges is harmless as long as it does not cause the merging of nodes, since we count multiplicities of edges using tokens and so no information can be lost in this way. In our case we consider spurious runs to be caused by the merging of graph nodes during the construction of the over-approximation.

If in the construction of an approximated unfolding we do not use coverability in order to control the construction of an over-approximating Petri graph (see Section 3.2, Construction 3.2.7), then we should demand additionally that in each rule r no two edges in the left-hand side L_r have the same label. Otherwise, if, for example, the left-hand side of the error rule consists of two edges labelled with "A", then this rule can also be spuriously unfolded in the case of a single edge labelled "A" in the Petri graph. This means that in this case it is not always the merged nodes which are the reason for the spurious run.

We believe that the technique of identifying the reason for the spurious run is independent of the abstraction mechanism used in this thesis and could also be used in other frameworks dealing with approximations of graph structures. The results of this section are published in [68].

5.1.1 Spurious Runs

Remember that we defined a real run \mathcal{J}_r as a sequence

$$(G_0 \Rightarrow_{r_1} G_1 \Rightarrow_{r_2} \dots \Rightarrow_{r_n} G_n)$$

of the GTS (\mathcal{R}, G_0) , which can be considered as an abstract run of the GTS (see Section 3.2, Definition 3.2.15) where the morphisms $\varphi_i : L_{i+1} \rightarrow G_i$ represent the matches of the left-hand sides of the rules r_i .

Let us remind here that given a hypergraph $graph(m)$ and a morphism $\psi : graph(m) \rightarrow G$ injective on nodes with $\psi^\oplus(E_H) = m$ and a morphism $\varphi : G' \rightarrow G$ such that $\varphi^\oplus(E_{G'}) \leq m$ there exists an edge-injective morphism $e_{m,\varphi} : G' \rightarrow graph(m)$ such that $\psi \circ e_{m,\varphi} = \varphi$. The morphism $e_{m,\varphi}$ is used below.

An abstract run of a Petri graph \mathcal{J} is a sequence

$$graph(m_0) \Rightarrow_{r_1} graph(m_1) \Rightarrow_{r_2} \dots \Rightarrow_{r_n} graph(m_n)$$

together with morphisms $\varphi_i = e_{m_i, \mu(t_{i+1})|_{L_{i+1}}} : L_{i+1} \rightarrow \text{graph}(m_i)$ and can also be considered as an abstract run of the GTS (see Section 3.2). The markings m_i here form a firing sequence $m_0 [t_1] m_1 [t_2] \dots m_{n-1} [t_n] m_n$ of the underlying Petri net.

For a given abstract run

$$\mathcal{J} = (\text{graph}(m_0) \Rightarrow_{r_1} \text{graph}(m_1) \Rightarrow_{r_2} \dots \Rightarrow_{r_n} \text{graph}(m_n))$$

of the Petri graph with morphisms $\varphi_i : L_{i+1} \rightarrow \text{graph}(m_i)$ we define \mathcal{H} to be the set of real runs corresponding to the prefixes of \mathcal{J} . Furthermore let \mathcal{H}_i be the set of hypergraphs reachable after i steps in a real run $\mathcal{J}_r \in \mathcal{H}$. It holds that $\mathcal{H}_0 = \{G_0\}$.

An abstract run \mathcal{J} is spurious if $\mathcal{H}_n = \emptyset$. If the run is spurious, there exists a k such that $\mathcal{H}_k \neq \emptyset$, but $\mathcal{H}_{k+1} = \emptyset$ (and therefore also $\mathcal{H}_l = \emptyset$ for $l > k$). It will be shown in the following how to construct a new refined over-approximation $\mathcal{C}'_{\mathcal{G}}$, which does not contain \mathcal{J} and some other spurious runs corresponding to \mathcal{J} .

Example: Let us consider again the example GTS from Section 3.1 (represented in Fig. 3.1). The first (coarsest) approximation for this example is represented in Fig 3.3. As we can see from the underlying Petri net the edge labeled "Error" is reachable. One can immediately fire the transition "Error". However the rule "Error" cannot be applied to the initial graph of the GTS (Fig. 3.1). This means that the counterexample leading to the edge labeled "Error" is spurious.

Let us consider as example the spurious abstract run corresponding to the firing of the transition "Error" in Fig. 5.1. In fact, there is no real run in the original GTS that corresponds to it. We obtain markings $m_0 = \{A : 1, B : 2, C : 1\}$ and $m_1 = \{B : 2, \text{Error} : 1\}$ and two graphs $\text{graph}(m_0)$, $\text{graph}(m_1)$ generated by these markings (see Fig. 5.1). One can see that—due to over-approximation and the merging of all nodes—the edges A and C are now connected. This is the reason why the transition "Error" can be applied in the over-approximation while this is not possible in the GTS. Hence the run $\text{graph}(m_0) \Rightarrow_{\text{Error}} \text{graph}(m_1)$ is spurious and no real run corresponds to it.

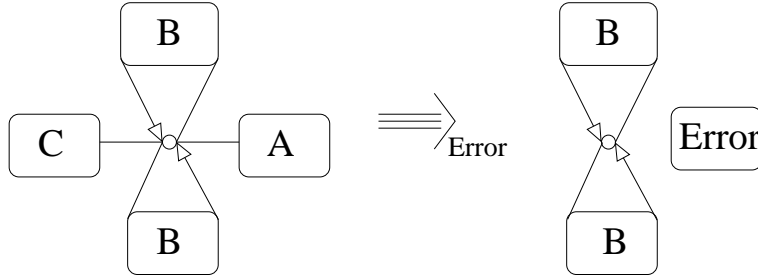


Figure 5.1: Left: $\text{graph}(m_0)$ generated by the initial marking. Right: $\text{graph}(m_1)$ where $m_0[\text{Error}]m_1$.

5.1.2 Relations on Nodes for Refining Abstract Runs

According to Algorithm 3.2.7 and Definitions 3.2.2 and 3.2.15 it holds that $\mathcal{H}_k \neq \emptyset$ and $\mathcal{H}_{k+1} = \emptyset$ if and only if for each $G \in \mathcal{H}_k$ there exists *no* edge-injective morphism $\eta : L_{k+1} \rightarrow G$ such that the following diagram commutes, where ξ_k is an edge-bijective morphism derived from the correspondence property (see Definition 3.2.15). In other

words: there is no way to find a match of the left-hand side in G that agrees with the abstract run.

$$L_{k+1} \xrightarrow{\eta} G \xrightarrow{\xi_k} \text{graph}(m_k)$$

φ_k

For if there were such a match morphism η , we could rewrite G to G' with rule r_{k+1} corresponding to the transition transforming m_k to m_{k+1} . Because of the construction of the Petri graph, where the right-hand side of r_{i+1} has been attached during an unfolding step, we would then be able to find an edge-bijective morphism $\xi_{k+1}: G' \rightarrow \text{graph}(m_{k+1})$ thus continuing the correspondence.

This means that rule r_{k+1} cannot be applied to any graph in \mathcal{H}_k such that the match of the left-hand side agrees with the match in the abstract run. It is easy to conclude that existence of the transition t_{k+1} means that the rule r_{k+1} can be applied to $\text{graph}(m_k)$. Morphism $\xi_k : G \rightarrow \text{graph}(m_k)$ is edge-bijective for each $G \in \mathcal{H}_k$. Such a situation is only possible if ξ_k is non-injective on some nodes of G , i.e., these nodes were merged during construction of the over-approximation \mathcal{C}_G , which is the reason for the spurious run.

Example: Fig. 5.2 shows the left-hand side of the rule "Error", the real graph contained in \mathcal{H}_0 , $\text{graph}(m_1)$ and the corresponding morphisms. Note that $\mathcal{H}_1 = \emptyset$, which corresponds to the fact that the run is spurious. Specifically there exists no morphism $L_1 \rightarrow G_0$ that makes the diagram commute.

The nodes v_1 , v_2 and v_3 of the initial hypergraph have been merged by the over-approximation. This led to the spurious abstract run depicted in Figure 5.1, which was obtained by firing the transition "Error" of the Petri net in Fig. 7.6.

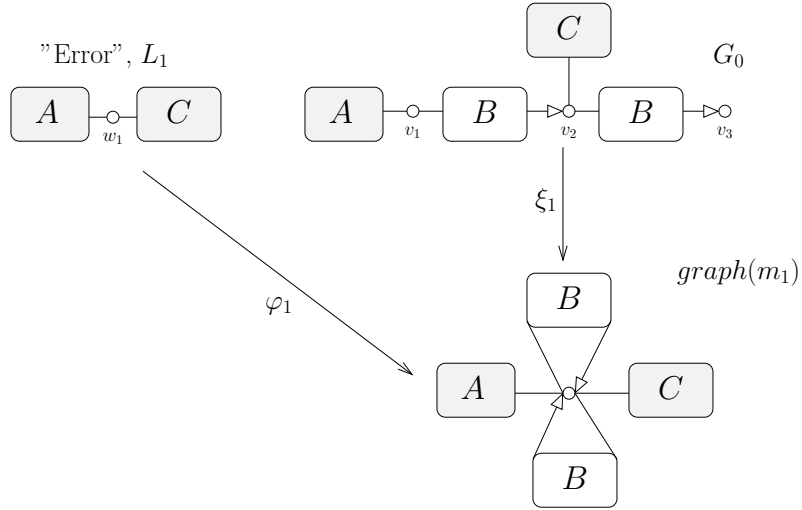


Figure 5.2: Left-hand side of the rule "Error", the real graph contained in \mathcal{H}_0 , $\text{graph}(m_1)$ and the corresponding morphisms.

We will now show how to determine the node merges which caused the spurious run. From the sets of such nodes we construct an equivalence relation on nodes and in the next section we show how this equivalence relation can be used in order to construct a better over-approximation without the given spurious run.

Consider, for a fixed graph G and a morphism ξ_k , the set Θ of possible equivalence relations \sim on nodes of a graph $G \in \mathcal{H}_k$ such that, after merging the nodes in each equivalence class, we can find an appropriate match of the left-hand side L_{k+1} in the

graph G/\sim . More formally, we demand the existence of an edge-injective morphism $\eta' : L_{k+1} \rightarrow G/\sim$ such that the following diagram commutes, where $\xi'_k : G/\sim \rightarrow \text{graph}(m_k)$ is obtained by quotienting ξ_k according to \sim .

$$\begin{array}{ccc} L_{k+1} & \xrightarrow{\eta'} & G/\sim \xrightarrow{\xi'_k} \text{graph}(m_k) \\ & \searrow \varphi_k & \nearrow \end{array}$$

In order to characterize the smallest equivalence in Θ , consider a node v of the left-hand side and determine a set Q_v of nodes in G which have to be fused into one node which is the image of v under η' . Let $v \in V_{L_{k+1}}$ and let e be an edge of L_{k+1} with¹ $c_i(e) = v$ for some i . For every edge e' in G with $\xi_k(e') = \varphi_k(e)$ we require that $c_i(e') \in Q_v$.

Consider the relation \mathcal{Q} , where for each $v \in V_{L_{k+1}}$ all nodes in Q_v are related and the relation $\widehat{\mathcal{Q}}$ which is the smallest equivalence containing \mathcal{Q} .

Proposition 5.1.1 *The equivalence $\widehat{\mathcal{Q}}$ constructed above is the smallest equivalence contained in Θ .*

Proof: First note that the set Q_v can also be defined as follows:

$$Q_v = \{c_i(\xi_k^{-1}(\varphi_k(e))) \mid e \in E_{L_{k+1}}, v = c_i(e)\}.$$

Let \widehat{Q}_v be the equivalence class in $\widehat{\mathcal{Q}}$ containing Q_v . Note that for each $v_1, v_2 \in L$: $\widehat{Q}_{v_1} \cap \widehat{Q}_{v_2} = \emptyset$ or $\widehat{Q}_{v_1} = \widehat{Q}_{v_2}$.

We assume that there is a morphism $\mu : G \rightarrow G/\widehat{\mathcal{Q}}$ mapping every edge to itself and every node $w \in G$ to the equivalence class \widehat{Q}_v , where $w \in \widehat{Q}_v$. Furthermore there is a morphism $\xi'_k : G/\widehat{\mathcal{Q}} \rightarrow \text{graph}(m_k)$ mapping every edge e to $\xi_k(\mu^{-1}(e))$ and every node \widehat{Q}_v to $\xi_k(w)$ where $w \in Q_v$. This is well-defined since $w_1 \mathcal{Q} w_2$ implies $\xi_k(w_1) = \xi_k(w_2)$ and $\widehat{\mathcal{Q}}$ is a transitive closure of \mathcal{Q} . Therefore $w_1 \widehat{\mathcal{Q}} w_2$ implies $\xi_k(w_1) = \xi_k(w_2)$.

Now let us define a morphism $\eta' : L_{k+1} \rightarrow G/\widehat{\mathcal{Q}}$ and show that it is a hypergraph morphism. Set $\eta'(v) = \widehat{Q}_v$ for each $v \in V_L$ and $\eta'(e) = \xi_k^{-1}(\varphi_k(e))$ for each $e \in E_L$. Since φ_k is injective on edges and ξ_k is bijective on edges, η' is also injective on edges. In order to show that η' is a morphism we have to prove that $\eta'(c_i(e)) = c_i(\eta'(e))$. Let $v = c_i(e)$. We have $\eta'(c_i(e)) = \widehat{Q}_v$. On the other hand $c_i(\eta'(e)) = c_i(\xi_k^{-1}(\varphi_k(e))) = c_i(\mu(\xi_k^{-1}(\varphi_k(e)))) = \mu(c_i(\xi_k^{-1}(\varphi_k(e)))) = \widehat{Q}_v$, since $c_i(\xi_k^{-1}(\varphi_k(e))) \in Q_v$.

We now have the following situation:

$$\begin{array}{ccccc} & & \xrightarrow{\varphi_k} & & \\ L_{k+1} & \xrightarrow{\eta'} & G/\widehat{\mathcal{Q}} & \xrightarrow{\xi'_k} & \text{graph}(m_k) \\ & \uparrow \mu & \uparrow \xi_k & & \\ & G & & & \end{array}$$

Next we show that the diagram above commutes, i.e., $\xi'_k \circ \eta' = \varphi_k$. By definition it commutes on edges. For nodes we show that $\xi'_k(\widehat{Q}_v) = \varphi_k(v)$ for each $v \in V_L$. Let $w \in Q_v$.

According to the definition $w = c_i(\xi_k^{-1}(\varphi_k(e)))$ for an edge e and an index i where $c_i(e) = v$.

We have $\xi'_k(\eta'(v)) = \xi'_k(\widehat{Q}_v) = \xi_k(w) = \xi_k(c_i(\xi_k^{-1}(\varphi_k(e)))) = \varphi_k(c_i(e)) = \varphi_k(v)$

¹Let us remind here that by $c_i(e)$ we denote the i -th node in the sequence $c(e)$.

We have proved that $\widehat{\mathcal{Q}} \in \Theta$. Now we show that $\widehat{\mathcal{Q}}$ is the smallest equivalence relation in Θ . Let $\widetilde{\mathcal{Q}}$ be another equivalence relation from Θ where $\tilde{\eta}: L_{k+1} \rightarrow G/\widetilde{\mathcal{Q}}$, $\tilde{\mu}: G \rightarrow G/\widetilde{\mathcal{Q}}$ and $\tilde{\xi}_k: G/\widetilde{\mathcal{Q}} \rightarrow \text{graph}(m_k)$ such that $\tilde{\xi}_k \circ \tilde{\eta} = \varphi_k$ and $\tilde{\xi}_k \circ \tilde{\mu} = \xi_k$.

Let $w_1 \mathcal{Q} w_2$ where $w_1, w_2 \in V_G$ and $w_1 \neq w_2$. That means that there are edges $e_1, e_2 \in E_L$ and indexes j_1, j_2 such that $w_i = c_{j_i}(\xi_k^{-1}(\varphi_k(e_i)))$ and there exists a node $v \in V_L$ such that $c_{j_i}(e_i) = v$ for $i = 1, 2$. It holds that

$$\begin{aligned} \tilde{\mu}(w_i) &= \tilde{\mu}(c_{j_i}(\xi_k^{-1}(\varphi_k(e_i)))) = c_{j_i}(\tilde{\mu}(\xi_k^{-1}(\varphi_k(e_i)))) = c_{j_i}(\tilde{\xi}_k^{-1}(\varphi_k(e_i))) \\ &= c_{j_i}(\tilde{\eta}(e_i)) = \tilde{\eta}(c_{j_i}(e_i)) = \tilde{\eta}(v) \end{aligned}$$

Note that $\hat{\xi}_k^{-1}(\varphi_k(e_i)) = \hat{\eta}(e_i)$ only holds since $\hat{\eta}, \varphi_k$ are injective on edges and $\hat{\xi}_k^{-1}$ is bijective on edges.

Hence we have $\tilde{\mu}(w_1) = \tilde{\mu}(w_2)$ and w_1, w_2 must be in the same equivalence class according to $\widetilde{\mathcal{Q}}$. This means $\mathcal{Q} \subseteq \widetilde{\mathcal{Q}}$. By definition $\widehat{\mathcal{Q}}$ is the the smallest equivalence relation containing \mathcal{Q} . Therefore $\widehat{\mathcal{Q}} \subseteq \widetilde{\mathcal{Q}}$. \square

Example: We consider again the abstract error trace \mathcal{J} which can be obtained by firing the transition “Error”. However, this error trace has no real runs that correspond to it, which can be seen by computing the set \mathcal{H} of runs corresponding to prefixes of \mathcal{J} . Here, the set \mathcal{H}_0 consists of the initial hypergraph and the rule “Error” cannot be applied to G_0 in such a way that the corresponding diagram commutes and therefore the set \mathcal{H}_1 is empty.

Fig. 5.2 shows the left-hand side of rule “Error”, $G_0 \in \mathcal{H}_0$ and $\text{graph}(m_0)$, the graph corresponding to the initial marking. One notices that no appropriate morphism $\eta: L_1 \rightarrow G_0$ can be found unless the nodes v_1 and v_2 in G_0 are merged. Therefore we have $Q_{w_1} = \{v_1, v_2\}$ and the smallest equivalence relation $\widehat{\mathcal{Q}}$ relates the nodes v_1 and v_2 and no other nodes.

Now we know how, for a given spurious run and an empty set \mathcal{H}_k , one can construct the smallest equivalence relation on the nodes of the hypergraph G such that if we merge the equivalent nodes in the hypergraph G (i.e., factor it through the equivalence relation), then the rule r_{k+1} can be applied to the obtained hypergraph and the corresponding diagram above commutes. In the next section we show how to use this equivalence relation in order to construct an over-approximation without the given spurious run.

5.1.3 Elimination of Spurious Runs

The general idea for destroying spurious runs is to avoid the merging of nodes from the same equivalence class of $\widehat{\mathcal{Q}}$. For this reason we assign colours to the nodes of the graphs contained in \mathcal{H} and disallow the merging of nodes corresponding to nodes with the same colour. For reasons that will become clear below a node may have several colours, i.e., a node v is associated to a set $\text{cols}(v)$ of colours.

For each $G \in \mathcal{H}_k$ and each morphism $\xi_k: G \rightarrow \text{graph}(m_k)$ we consider the corresponding relation \mathcal{Q}_{G, ξ_k} . Then we assign colours to nodes in such a way that there exists at least one pair v_1, v_2 of nodes such that $v_1 \mathcal{Q}_{G, \xi_k} v_2$ and $\text{cols}(v_1) \cap \text{cols}(v_2) \neq \emptyset$. There are several ways to do this and all of them will help to eliminate the counterexample. In our implementation we choose a color for each set of nodes Q_v and assign it to all nodes contained in Q_v .

In order to catch “bad” mergings as early as possible, these colours have to be distributed to the remaining graphs contained in \mathcal{H} . Let us recall here that according to

Definition 3.1.3 for each real run $\mathcal{J}_r = (G_0 \Rightarrow_{r_1} G_1 \dots \Rightarrow_{r_k} G_k)$ from \mathcal{H} we have injective partial morphisms $\nu_i : G_i \rightarrow G_{i+1}$ for $i = 0, \dots, k-1$. Using these partial morphisms we assign the colours of G_k to the remaining graphs G_i contained in \mathcal{H} . We start from G_k and proceed as follows: if a node $v \in G_{i+1}$ has a colour then we also assign this colour to the node $\nu^{-1}(v)$ if such a node exists. In this way a node may obtain several colours, due to the branching structure of the runs contained in \mathcal{H} . We denote by $cols(v)$ the set of colours of the node $v \in V_{G_j}$ where $G_j \in \mathcal{H}_j$.

We are now ready to present the algorithm for computing the refined over-approximation.

Algorithm 5.1.2 (Refined approximated unfolding)

Input: A GTS \mathcal{G} , a set \mathcal{H} of runs corresponding to prefixes of the counterexample and a function $cols$ assigning sets of colours to the nodes of the graphs in \mathcal{H} .

Output: The refined over-approximation $\mathcal{C}'_{\mathcal{G}}$.

We start constructing the new over-approximation $\mathcal{C}'_{\mathcal{G}}$ with the initial graph G_0 . Unfolding steps will be performed as described in Algorithm 3.2.7.

For a folding step we disallow the merging of nodes corresponding to nodes in \mathcal{H} having the same colour. More specifically, consider the over-approximation $\mathcal{C}'_{\mathcal{G}}$, which is currently being constructed. Now for each run $\mathcal{J}_r = G_0 \Rightarrow_{r_1} \dots \Rightarrow_{r_\ell} G_\ell$ in \mathcal{H} where $\ell < k$ check the following:

We consider all abstract runs $\mathcal{J} = graph(m_0) \Rightarrow_{r_1} \dots \Rightarrow_{r_\ell} graph(m_\ell)$ of the current Petri graph $\mathcal{C}'_{\mathcal{G}}$ for which $\mathcal{J}_r \ll \mathcal{J}$ and all edge-bijective morphisms $\xi: G_i \rightarrow graph(m_i)$ for $i = 0, \dots, \ell$. Whenever there are two nodes v_1, v_2 in G_i with $cols(v_1) \cap cols(v_2) \neq \emptyset$ and $\xi(v_1) = \xi(v_2)$, we have erroneously merged two nodes in the approximation which should not have been merged. Consequently this folding step is undone.

Previously rejected folding steps are recorded and are not any more considered by the algorithm.

In this way we will eliminate not only the spurious run but several more runs which are characterized below (see Proposition 5.1.6). All these abstract runs correspond to the original spurious run and have a weak correspondence to a prefix of some run in \mathcal{H} of maximal length. We say in this case that the eliminated runs correspond to the spurious run with respect to \mathcal{H} .

We have decided to check abstract runs against all runs in \mathcal{H} having maximal length. This decision is somewhat arbitrary but works well in practice. In order to make the spurious run disappear it is enough to compare with at least one of the runs in \mathcal{H} .

The current implementation checks all abstract runs to which the prefix of at least one run in \mathcal{H} corresponds. However, the theory would work just as well if we considered (strong) correspondence at this point.

Before continuing with the running example, the following two remarks are in order: First note that the check in Algorithm 5.1.2 which is performed for each folding step can be done in an efficient way by following the branching structure of the runs instead of enumerating all runs. Second, refining the abstraction directly without constructing it again from scratch as we have done, is a non-trivial undertaking since Petri graphs are very compact descriptions of the state space in which states can not be easily separated. Doing this in an efficient way is a direction of future work.

Example: Fig. 5.3 depicts the Petri graph obtained for our running example in Fig. 3.1 after the abstraction refinement procedure. As one can see, the “critical nodes” of the hypergraph, namely the nodes v and w , are now separated and the edge “Error” is no more present. This means that we have successfully verified the example.

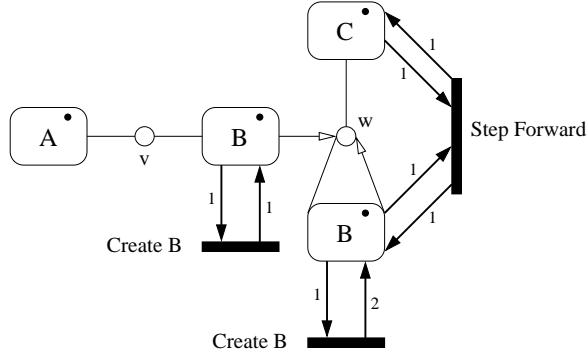


Figure 5.3: Refined Petri graph

5.1.4 Correctness

In the following we will show that Algorithm 5.1.2 terminates and that the refined over-approximation is correct and more exact than the previous one.

Let \mathcal{C}_G be an over-approximation with a spurious run \mathcal{J} and let \mathcal{C}'_G be the corresponding refined over-approximation. In [6] it is shown that the algorithm constructing the over-approximating Petri graph terminates. We modified the algorithm by forbidding some of the folding steps and hence we have to reprove termination for the new version of the algorithm.

Proposition 5.1.3 *The algorithm computing the refined over-approximation \mathcal{C}'_G for a given GTS \mathcal{G} and a (spurious) abstract run \mathcal{J} of \mathcal{C}_G terminates.*

Proof: (Sketch) By a slight modification of the termination proof for the approximated unfolding algorithm in [4].

We can show termination by modifying the termination proof that shows termination for the computation of k -coverings. Let n be the length of the spurious run that is to be avoided. Then, assign different labels to transitions the merging of which would cause nodes corresponding to coloured nodes in \mathcal{H} to be merged. This can be done by introducing mappings $d(t): V_L \rightarrow V_G \cup \{\perp\}$ where L is the left-hand side of the rule associated with t and G is the graph underlying the Petri graph. To every node which does not correspond to a coloured node in \mathcal{H} we assign \perp , to all other nodes we assign the corresponding node in G . Since the net is irredundant and all nodes in question are generated after at most n steps, only finitely many nodes can be involved and hence there are only finitely many different mappings $d(t)$.

The colours introduced for the nodes in \mathcal{H} cannot be used, since they have the opposite function than the one required here: in this case nodes must *not* be merged if they have the same colour.

The rest of the termination proof can be carried out as before. □

Furthermore the new over-approximation is still a valid over-approximation as before.

Proposition 5.1.4 *Let \mathcal{C}'_G be the refined over-approximation of the GTS. Then, for every real run \mathcal{J}_r of the graph transformation system there exists an abstract run $\mathcal{J} \in \text{Run}_A(\mathcal{C}'_G)$ such that \mathcal{J}_r corresponds to \mathcal{J} , i.e., $\mathcal{J}_r \lll \mathcal{J}$.*

Proof: This follows directly from the construction of the over-approximation, since the construction starts with the initial graph and every coverable left-hand side is unfolded at some point. \square

In the following two propositions we will show that we have eliminated the given spurious counterexample and have not added any new ones. First we should answer the following question: what kind of runs have we eliminated by abstraction refinement? It is easy to see that in the refined over-approximation we have lost the initial spurious counter-example \mathcal{J} . In fact we have not only eliminated \mathcal{J} , but some more runs as described below.

Definition 5.1.5 (Correspondence with respect to runs) *Let (P, ι) and (P', ι') be two Petri graphs for a GTS (\mathcal{R}, G_0) . Furthermore let $\mathcal{J} \in \text{Run}_A(P, \iota)$ and $\mathcal{J}' \in \text{Run}_A(P', \iota')$ be two abstract runs of these Petri graphs and let \mathcal{H} be the set of real runs considered earlier. We say that \mathcal{J}' corresponds to \mathcal{J} with respect to \mathcal{H} if \mathcal{J}' corresponds to \mathcal{J} and a run $\mathcal{J}'' \in \mathcal{H}$ of maximal length weakly corresponds to a prefix of \mathcal{J}' , i.e., $\mathcal{J}' \lll \mathcal{J}$ and $\mathcal{J}'' \ll pr_{|\mathcal{J}''|}(\mathcal{J}')$ for some $\mathcal{J}'' \in \mathcal{H}$. (By $pr_\ell(\mathcal{J})$ we denote the prefix of length ℓ of a run \mathcal{J} .)*

As specified in the definition, we say that an abstract run \mathcal{J}' corresponds to an abstract run \mathcal{J} (which further will be a spurious run to be eliminated) with respect to \mathcal{H} if \mathcal{J}' corresponds to \mathcal{J} and at the same time has a weak correspondence to a prefix of some run in \mathcal{H} of maximal length. Using this definition we can now state and prove the following propositions.

Proposition 5.1.6 *The refined over-approximation \mathcal{C}'_G , constructed above does not contain any run \mathcal{J}' corresponding to the spurious run \mathcal{J} of \mathcal{C}_G with respect to \mathcal{H} .*

Proof: Let us consider \mathcal{J}' corresponding to the spurious run \mathcal{J} of \mathcal{C}_G with respect to \mathcal{H} and let k be the maximal index such that \mathcal{H}_k is not empty. Let $\mathcal{J} = \text{graph}(m_0) \Rightarrow_{r_1} \dots \Rightarrow_{r_n} \text{graph}(m_n)$ and $\mathcal{J}' = \text{graph}(m'_0) \Rightarrow_{r_1} \dots \Rightarrow_{r_n} \text{graph}(m'_n)$.

We consider the following diagram where L_{k+1} is the left-hand side of the rule r_{k+1} and $G_k \in \mathcal{H}_k$ is a reachable hypergraph from a maximal run \mathcal{J}'' weakly corresponding to a prefix of \mathcal{J}' ($\mathcal{J}'' \ll pr_{|\mathcal{J}''|}(\mathcal{J}')$).

$$\begin{array}{ccccc}
 & & \varphi_k & & \\
 & \nearrow & & \searrow & \\
 L_{k+1} & \xrightarrow{\varphi'_k} & \text{graph}(m'_k) & \xrightarrow{\xi} & \text{graph}(m_k) \\
 & \uparrow \psi' & & \nearrow \psi & \\
 & G_k & & &
 \end{array}$$

The existence of ψ' is implied by the weak correspondence $\mathcal{J}'' \ll pr_{|\mathcal{J}''|}(\mathcal{J}')$. The sub-diagram $\{L_{k+1}, \text{graph}(m'_k), \text{graph}(m_k)\}$ is taken from the condition $\mathcal{J}' \lll \mathcal{J}$. We define $\psi = \xi \circ \psi'$. This means that the sub-diagram $\{G_k, \text{graph}(m'_k), \text{graph}(m_k)\}$ commutes.

The morphism ψ' is edge-bijective. This implies that $\text{graph}(m'_k)$ is isomorphic to the graph G_k factorized through an equivalence \sim on nodes. This equivalence \sim must be an element of the set of equivalences Θ defined in Section 5.1.2 of which $\hat{\mathcal{Q}}_{G_k, \psi}$ is the smallest element according to Proposition 5.1.1. This implies $\sim \supseteq \mathcal{Q}_{G_k, \psi}$ and we can conclude that ψ' maps at least two nodes having the same colour to the same node.

Hence we have a situation where $\mathcal{J}'' \ll pr_{|\mathcal{J}''|}(\mathcal{J}')$ and there is a morphism $\psi' : G_k \rightarrow \text{graph}(m'_k)$ with $\psi'(v_1) = \psi'(v_2)$, where $v_1 \neq v_2$ and $\text{cols}(v_1) \cap \text{cols}(v_2) \neq \emptyset$.

This is a contradiction, since this situation should have been detected by the abstraction refinement algorithm. \square

We can also show that no new runs have appeared, which means that the new approximation is strictly better than the old one. In particular if a run \mathcal{J}' from \mathcal{C}'_G leads to some error graph $\text{graph}(m'_e)$, then the corresponding \mathcal{J} leads to the graph $\xi(\text{graph}(m'_e))$. The proof of the following proposition is based on the categorical description of GTSSs. A brief introduction to category theory and its applications to the graph rewriting can be found in Appendix A. The definition of a category theory can be found in [3, 73, 22] and its applications to graph rewriting can be found in [95, 12].

Proposition 5.1.7 *If the refined over-approximation \mathcal{C}'_G contains a run \mathcal{J}' , then it corresponds to some run \mathcal{J} in \mathcal{C}_G .*

Proof: This follows from the fact, that from the Petri graph P' of the refined approximation there exists a morphism $\beta : P' \rightarrow P$ to the original Petri graph P , which can be shown by induction on the number of steps of the algorithm.

We construct sequences of Petri graphs $P_i = (G_i, N_i, \mu_i)$ and $P'_i = (G'_i, N'_i, \mu'_i)$ in parallel. In both cases we start with the initial graph G_0 . There exists a trivial morphism (the identity) $\beta_0 : G_0 \rightarrow G_0$. After n we obtain the partial Petri graphs P_n and P'_n and a morphism $\beta_n : P'_n \rightarrow P_n$.

Now there are several possible cases. Let us assume that there is a match $\varphi'_n : L \rightarrow G'_n$ of the left hand side in P'_n which can be unfolded by adding transition t' to P'_n . Then $\beta_n \circ \varphi'_n$ is also a match for P_n . If it can be unfolded as well, we extend P_n accordingly by adding a transition t and β_n can be extended to $\beta_{n+1} : P'_{n+1} \rightarrow P_{n+1}$ by setting $\beta_{n+1}(t') = t$ and $\beta_{n+1}(t'^\bullet) = t^\bullet$.

This unfolding step is disallowed for P_n , because there exists a folding possibility i.e., the folding step can be made with some transition t in P_n such that $p_{N_n}(t) = p_{N'_n}(t')$ and $\beta_n \circ \varphi'_n$ is the first match of the left-hand side for the folding step. We obtain the following diagram. On the right the pushout corresponding to the unfolding step in P'_n is depicted, i.e., ν', φ' are obtained as the pushout of η, φ'_n , and on the left the coequalizer corresponding to the folding step in P_n is depicted, i.e., ν is the coequalizer of $\beta_n \circ \varphi'_n$ and $\psi \circ \eta$. The morphism $\beta_n : P'_n \rightarrow P_n$ is assumed to exist by the induction hypothesis. We set $\varphi = \nu \circ \psi$.

It holds that $\nu \circ \beta_n \circ \varphi'_n = \nu \circ \psi \circ \eta = \varphi \circ \eta$, hence the morphism β_{n+1} exists as mediating morphism of the pushout.

In the remaining case a folding step is made in P'_n . In this case either the images of the two left-hand sides under β_n are already equal (in this case β_{n+1} by factorizing of β_n) or we can perform the same folding step in P_n , since the conditions allowing folding steps are preserved by morphisms.

Hence we obtain the following diagram, where ν is the coequalizer of η_1, η_2 and ν' is the coequalizer of η'_1, η'_2 .

$$\begin{array}{ccccc}
 L & \xrightarrow{\eta'_1, \eta'_2} & P'_n & \xrightarrow{\nu'} & P'_{n+1} \\
 & \searrow \eta_1, \eta_2 & \downarrow \beta_n & & \downarrow \beta_{n+1} \\
 & & P_n & \xrightarrow{\nu} & P_{n+1}
 \end{array}$$

It holds that $(\nu \circ \beta_n) \circ \eta'_1 = \nu \circ \eta_1 = \nu \circ \eta_2 = (\nu \circ \beta_n) \circ \eta'_2$, hence the mediating morphism of the coequalizer gives us a morphism $\beta_{n+1}: P'_{n+1} \rightarrow P_{n+1}$.

Notice that the previous case, where no folding is necessary in P_n , can be considered as a sub-case of this one by setting $\eta_1 = \eta_2$ and $\nu = id$.

After the unfolding and folding of P'_n terminates, this might not be the case of P_n . Then, continue by working on P_n , producing a sequence of morphisms $P_n \xrightarrow{\psi_{n+1}} \dots \psi_m \rightarrow P_m$, where the algorithm terminates after P_m has been constructed. Then the morphism $\beta = \psi_m \circ \dots \circ \psi_{n+1} \circ \beta_n: P' \rightarrow P$ is the desired Petri graph morphism. \square

We remark that the considered abstraction refinement approach can also be implemented in the case of any number of spurious counterexamples as follows: We store the set \mathcal{H} with the internal structure and check the obtained over-approximation. If again a counterexample is found and it is spurious, then we apply the algorithm above to the new set \mathcal{H}' and the set \mathcal{H} , obtained in the previous step. This procedure can be repeated. Naturally, due to undecidability and the fact that GTSs are in general Turing-complete, there is no guarantee that it will ever terminate.

We have shown which spurious runs are eliminated in the refined abstraction. Among these is the initial spurious run \mathcal{J} . It is also shown that this procedure cannot produce new spurious runs. Hence the refined approximation $\mathcal{C}'_{\mathcal{G}}$ is better than the initial over-approximation $\mathcal{C}_{\mathcal{G}}$.

The counterexample-guided abstraction refinement technique described in this section is implemented in the tool Augur 2 (see Section 6). The tool can also work with several spurious counterexamples, which is important if the refinement process is iterated. The experimental results (Section 7) show effectiveness of the developed refinement approach.

5.2 Abstraction Refinement of Attributed Graph Transformation Systems

This section generalizes the abstraction refinement technique from Section 5.1. We develop here the technique of abstraction refinement for attributed graph transformation systems. As it was described in Section 4.4 we approximate AGTSs by attributed Petri graphs. In this case there are two possibilities to obtain a spurious counterexample: The first reason for the appearance of a spurious counterexample can be the merging of nodes (in the way it is described in the previous section). In this case structural refinement similar to the one described above should be applied. The second possibility can be the too coarse approximation of attributes (approximation of attributes to finite domains is necessary in order to obtain an analyzable over-approximation). In this case we refine the abstraction of attributes in the way it is described in Section 4.1. The main focus of this chapter is on deciding whether structural or attribute abstraction should be refined. The results of this section are published in [70].

We start abstraction refinement with an attributed Petri graph \mathcal{P}^a which is obtained by unfolding an AGTS \mathcal{G} and interpreting the resulting Petri graph in \mathcal{B} (as described in

Section 4.4). If the property we want to verify is violated, we obtain a counterexample of the following form:

$$\hat{m}_0[t_1, \hat{h}_1] \dots [t_n, \hat{h}_n] \hat{m}_n,$$

where the t_i are transitions and the \hat{h}_i are the corresponding bindings. Usually the AGTSs that we consider have an error rule and the property we want to verify is that this rule is not applicable. Hence a spurious run will usually include a firing of this error rule.

This counterexample can be seen as an abstract run with abstracted attributes of the following form:

$$\hat{\mathcal{J}}_A = (\text{graph}(\hat{m}_0) \Rightarrow_{r_1} \text{graph}(\hat{m}_1) \Rightarrow_{r_2} \dots \Rightarrow_{r_n} \text{graph}(\hat{m}_n)),$$

where $r_j = p_N(t_j)$ and $(\nu_j, \hat{h}_j) : L_j \rightarrow \text{graph}(\hat{m}_{j-1})$ are the corresponding morphisms from the left-hand side of r_j to $\text{graph}(\hat{m}_{j-1})$ for $j = 1, \dots, n$.

After analysing the Petri graph P^a we have the following four possibilities:

1. The property is successfully verified. This means that no counterexample was found in P^a . According to Proposition 4.4.5 we have also verified the property in P and according to Proposition 4.4.4 the property holds also in the original AGTS \mathcal{G} .
2. A real (non-spurious) counterexample $\hat{\mathcal{J}}_A$ as specified above is found. That is, we have $\mathcal{J}_R \lll \hat{\mathcal{J}}_A$ for a real run \mathcal{J}_R of \mathcal{G} (see Fig. 5.4). In this case we have found an error.

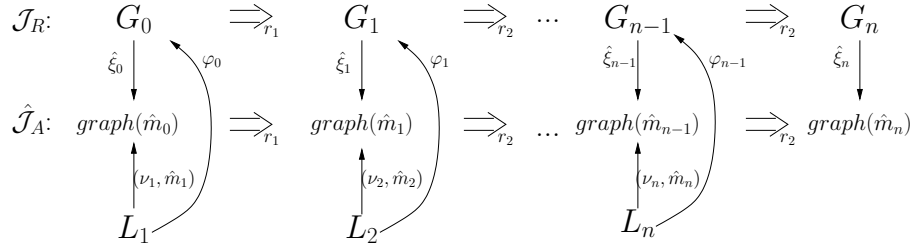


Figure 5.4: Real counterexample

3. The detected counterexample is *spurious*. This means that no real run \mathcal{J}_R with $\mathcal{J}_R \lll \hat{\mathcal{J}}_A$ exists.

However, there could be real runs \mathcal{J}'_R shorter than $\hat{\mathcal{J}}_A$ that correspond to a prefix $pr_i(\hat{\mathcal{J}}_A)$ of the counterexample, i.e., $\mathcal{J}'_R \lll pr_i(\hat{\mathcal{J}}_A)$. Let k be the maximal length of a real run corresponding to a prefix of $\hat{\mathcal{J}}_A$. The set of all such maximal real runs (there could be several of them) is denoted by \mathcal{H} .

For a given $\mathcal{J}'_R \in \mathcal{H}$ there always exists a (unique) run \mathcal{J}'_A of the attributed Petri graph P (with concrete attributes) with morphisms $(\nu_j, h_j) : L_j \rightarrow \text{graph}(m_{j-1})$ (morphisms ν_j as above) such that $\mathcal{J}'_R \lll \mathcal{J}'_A$ (see Fig. 5.5). It is easy to see that also $\mathcal{J}'_A \lll pr_i(\hat{\mathcal{J}}_A)$.

We now distinguish the following two cases:

- (a) We say that the over-approximation is *structurally too coarse* if for some $\mathcal{J}'_R \in \mathcal{H}$ the corresponding run \mathcal{J}'_A can be extended to a run \mathcal{J}''_A of length $k+1$ with a morphism $(\nu_{k+1}, h_{k+1}) : L_{k+1} \rightarrow \text{graph}(m_k)$ in such a way that $\mathcal{J}''_A \lll pr_{k+1}(\hat{\mathcal{J}}_A)$. The set of such prefixes from \mathcal{H} is denoted by \mathcal{H}_S .

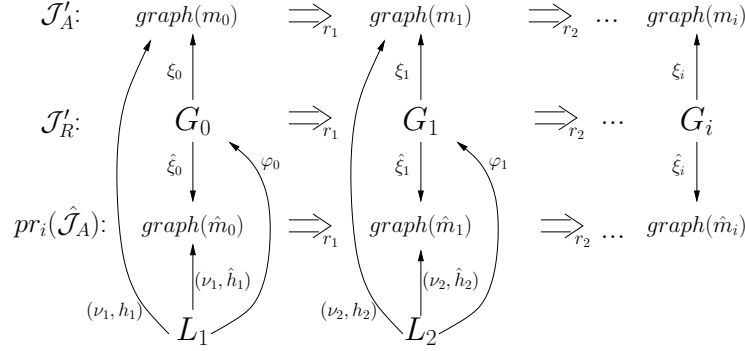


Figure 5.5: Counterexample (abstract and real runs with corresponding left-hand sides)

Both \mathcal{J}'_R and \mathcal{J}'_A have concrete attributes (which are not abstracted). However the run \mathcal{J}'_A can be extended to length $k + 1$, whereas the run \mathcal{J}'_R cannot be correspondingly extended. Because the attributes are not abstracted the reason for the appearance of the spurious counterexample is in the structural part of the Petri graph. In this case we can reduce the problem to the one we have described in the previous section.

Below we describe a technique based on the one proposed in Section 5.1, which allows us to eliminate the obtained counterexample if the over-approximation is structurally too coarse.

- (b) In the last case for each run $\mathcal{J}'_R \in \mathcal{H}$ such that $\mathcal{J}'_R \ll_{\sqsubseteq} pr_k(\hat{\mathcal{J}}_A)$, the corresponding run \mathcal{J}'_A can not be extended as in the previous case, i.e., $\mathcal{H}_S = \emptyset$. If this holds then we say that in the over-approximation P^a the attribute abstraction is too coarse i.e., the counterexample is spurious from the point of view of attributes.

As in the previous case, both \mathcal{J}'_R and \mathcal{J}'_A have concrete attributes (which are not abstracted). This means that without attribute abstraction we can reproduce in the over-approximating Petri graph as many steps (in the runs of maximal length k) as in the analyzed AGTS. This shows that the reason for the appearance of the spurious counterexample (of length at least $k + 1$) cannot be structural and is located in the abstraction of attributes which is too coarse. How we handle this case is also described below.

In the first two cases we have successfully solved the problem. If the obtained over-approximation is structurally too coarse and does not allow us to verify the property, a counterexample-guided abstraction refinement technique (Section 5.1) for refining the approximation is available. It starts from a concrete counterexample found by coverability checking of the Petri net and refines the structure of the Petri graph. In order to apply the technique described in Section 5.1 for non-attributed GTSS we consider the following diagram for each run of \mathcal{H}_S , where φ_k does not exist and the morphism ν_{k+1}, ξ_{k+1} are as in the diagram above.

$$\begin{array}{ccccc}
 L_{k+1} & \xrightarrow{\varphi_k} & G_k & \xrightarrow{\xi_k} & graph(m_k) \\
 & \searrow & & \nearrow & \\
 & & \nu_{k+1} & &
 \end{array}$$

As can be seen from the diagram above all morphisms are non-attributed and the diagram is exactly the one considered in Section 5.1.2. This means that we can apply

the refinement technique developed in Sections 5.1 without any changes to the case of attributed AGTSSs. The technique consists of determining the node merges which caused the spurious run and constructing a refined over-approximating Petri net (using the refined approximated unfolding from Section 5.1.3). The correctness of this approach in the case of attributed GTSSs will be proved below.

In this way, as in Section 5.1, we will eliminate not only the spurious run $\hat{\mathcal{J}}_A$ but all other abstract runs corresponding to it and at the same time having a weak correspondence to some run in \mathcal{H}_S (see Proposition 5.2.1 below). It can also be shown that no new spurious runs will be added. Furthermore the termination result for the non-attributed case (see Section 5.1.4) applies since attributes are added to the Petri graph only at the end of the unfolding procedure.

Proposition 5.2.1 *The structurally refined Petri graph P'^a constructed above (as in case 3(a)) does not contain any run $\hat{\mathcal{J}}'_A \sqsubseteq$ -corresponding to the spurious run $\hat{\mathcal{J}}_A$ of P^a and having a weak \sqsubseteq -correspondence to some run in \mathcal{H}_S .*

Furthermore if P'^a contains a spurious run $\hat{\mathcal{J}}'_A$, then it \sqsubseteq -corresponds to some run $\hat{\mathcal{J}}_A$ in P^a .

Proof: In Section 5.1 it is shown that for a structurally refined Petri graph P' without attributes it is not possible to obtain any run \mathcal{J}' corresponding to the spurious run $\hat{\mathcal{J}}_A$ without attributes and having a weak (also non-attributed) correspondence to some run in \mathcal{H}_S .

This means that in our case it is also not possible to construct such a run $\hat{\mathcal{J}}'_A$ even without taking attributes into consideration.

Also in Section 5.1 it is shown that we have a Petri graph morphism $\beta : P' \rightarrow P$, where P' and P are P'^a and P^a without considering attributes. A Petri graph morphism β is a morphism between hypergraph components of P' and P together with a mapping between transitions of P' and P preserving the initial markings and additionally presets and post-sets of each transition. The existence of such a morphism can be shown by induction on the number of steps of the algorithm.

If we perform structural refinement and refine a Petri graph P^a to P'^a we do not change the algebra. Hence the morphism β preserves the initial marking and every run $\hat{\mathcal{J}}'_A$ in P'^a can be simulated by a run $\hat{\mathcal{J}}_A$ in P^a . Simulation means correspondence and since the algebras coincide we have that $\hat{\mathcal{J}}'_A$ corresponds to $\hat{\mathcal{J}}_A$. □

Another possibility to eliminate the spurious counterexample is to use depth-based refinement (see Section 3) which constructs an over-approximation exact up to a pre-defined depth of the unfolding. Counterexample-guided abstraction refinement however usually results in smaller approximations and faster verification.

To refine the approximation in the last case we make our abstraction of attributes more exact. For example for the modulo abstraction we can increase the modulo base (we usually multiply it by two), and for the interval abstraction we can increase the numbers m and/or n . However in this case we have no guarantee that the spurious counterexample will be eliminated. We can, for example, refine the attributes a predefined number of times and after this, if the spurious counter-example is still reproducible, then we terminate the verification procedure with the answer “don’t know”. Future work in this case is the development and the implementation of the *predicate abstraction* technique [49, 41, 50, 59] for Petri graphs and AGTSSs. This technique uses predicates as an abstraction of data types and allows one to choose the necessary number of predicates in order to eliminate the given counterexample. The technique of predicate abstraction was developed for the standard program languages and transferring it to the Petri net

semantic is a non-trivial task. The difficulty of doing this will be further discussed in the conclusion.

So our results for the refinement of attribute abstraction refinement are definitely weaker than in the case of structure refinement. We can however still show that whenever we refine the attribute abstraction in a certain way, then no new spurious runs will appear.

Proposition 5.2.2 *Let $(\alpha_s : \mathcal{A}_s \rightarrow \mathcal{B}_s, \gamma_s : \mathcal{B}_s \rightarrow \mathcal{A}_s)_{s \in \mathcal{S}}$ be the Galois connection between algebras \mathcal{A}, \mathcal{B} which was originally used for attribute abstraction.*

Now let $(\alpha'_s : \mathcal{A}_s \rightarrow \mathcal{D}_s, \gamma'_s : \mathcal{D}_s \rightarrow \mathcal{A}_s)_{s \in \mathcal{S}}$ be a new connection from \mathcal{A} to \mathcal{D} .

We furthermore assume that there exists a Galois connection from \mathcal{D} to \mathcal{B} with mappings α''_s, γ''_s such that $\alpha_s \sqsubseteq \alpha''_s \circ \alpha'_s$. Then if the refined Petri graph P'^a contains a run $\hat{\mathcal{J}}'_A$, it \sqsubseteq -corresponds (with α''_s as \sqsubseteq -homomorphisms) to some run $\hat{\mathcal{J}}_A$ in P^a .

In particular, if $\hat{\mathcal{J}}'_A$ leads to a marking covering an error edge \hat{m}'_e of P'^a , then the corresponding run $\hat{\mathcal{J}}_A$ leads to a marking $\hat{m}_e \sqsubseteq \alpha''(\hat{m}'_e)$ that also covers the error edge.

Proof: Let $\hat{\mathcal{J}}'_A = (\hat{m}'_0 \Rightarrow_{r_1} \dots \Rightarrow_{r_n} \hat{m}'_n)$ be a spurious run in P'^a . For the initial markings we obtain $\alpha'(m_0) = \hat{m}'_0$, where m_0 is the initial marking of P and therefore $\alpha''(\hat{m}'_0) \sqsubseteq \hat{m}_0$, where $\hat{m}_0 = \alpha(m_0)$ is the initial marking in P^a .

Then, according to Proposition 4.4.5 there exists a run $\hat{\mathcal{J}}_A = (\hat{m}_0 \Rightarrow_{r_1} \dots \Rightarrow_{r_n} \hat{m}_n)$ such that $\hat{\mathcal{J}}'_A \ll \sqsubseteq \hat{\mathcal{J}}_A$. □

The abstraction refinement approach can also be implemented in the case of an arbitrary number of spurious counterexamples. This is needed whenever abstraction refinement is iterated. In this case we save the current state of the attribute abstraction and also store the set \mathcal{H}_S with the internal structure each time we make a structural refinement. Each time we construct a refined Petri graph we use the latest abstraction of attributes and all sets \mathcal{H}_S obtained in the previous iterations. This procedure can be repeated.

Example: Let us consider now the Petri graph in Fig. 4.7 using a modulo abstraction with base one (unit abstraction). The edge labelled Error of the Petri graph can be covered by firing transition “Error”. This means that either the property does not hold or the over-approximation is too coarse. One can show that the run is spurious, i.e., it has no counterpart in the original AGTS and the over-approximation is structurally too coarse. Applying abstraction refinement gives us a refined Petri graph, which is depicted in Fig. 5.6.

The edge labelled Error is still in the approximation and a counter-example still can be constructed (“Cross Backward”, “Error”). However, this counter-example can not be reproduced without approximation of attributes, which means that the attribute abstraction is too coarse and should be refined. By considering base two in the modulo abstraction we obtain a Petri graph in which the Error-edge is no longer coverable. This implies successful verification of the example.

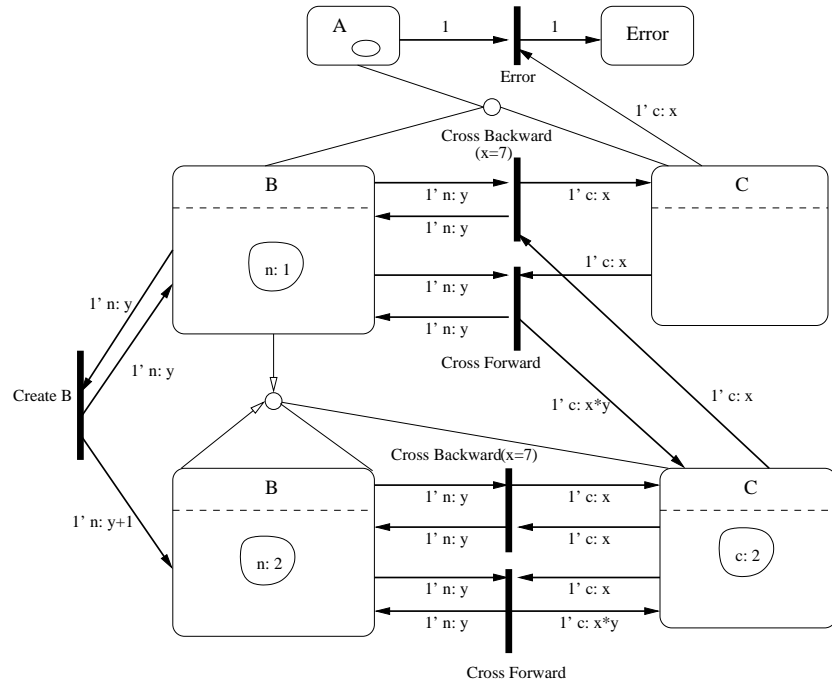


Figure 5.6: Petri graph after counterexample-guided abstraction refinement.

Chapter 6

Augur – a Tool for the Analysis of Graph Transformation Systems

In this chapter we describe the software environment AUGUR developed for the verification of GTSs and AGTSs. The environment consists of a previous version of the tool (AUGUR 1) and a new software tool called AUGUR 2. The design and implementation of AUGUR 2 is an important achievement of this thesis. We discuss here the problems and the demands which have led to the necessity of a completely new version of the tool and the background concepts of the software design of the new tool. With a new design we have tried to obtain a flexible and easily extendable software environment. We describe the main components and modules of the tool and also its functionality and usage. The results of this chapter have been published in [66, 67].

6.1 Brief Description of Augur 1

In the years from 2002 to 2006 the verification tool AUGUR 1 has been developed [66] which analyzes graph transformation systems (GTSs) by approximating them with Petri graphs¹. Using this tool several case studies have been already conducted, verifying, for instance, a mobile system with a firewall [7], a mutual exclusion protocol [33] and the insertion of elements into red-black trees [5].

The development of AUGUR 1 started with a small tool that reads GTXL files, constructs an approximating unfolding of the given GTS (as discussed in Chapter 3) and writes it in GXL files (GXL respectively GTXL are XML standards for the encoding of graphs and graph transformation systems [72]). Afterwards during the development of the tool the constant necessity of adding new features and new functionality has been faced. More specifically, the following components were added: analysis algorithms for Petri nets [106] based on coverability graphs [85] and backward reachability [1] (discussed in Chapter 2), an interface to Graphviz² for visualization purposes, the possibility to specify forbidden paths in graphs using regular expressions [86], the finite complete prefix technique for graph transformation systems [9, 16], the extension of the tool in order to use it for the purpose of test case generation [56]. Probably the most extensive addition was to

¹The tool can be obtained from http://www.ti.inf.uni-due.de/research/augur_1/.

²<http://www.graphviz.org/>

add support for counterexample-guided abstraction refinement³ (discussed in Chapter 5).

The architecture of AUGUR was strongly oriented towards the concrete task of computing approximated unfoldings of GTSS. This made all changes mentioned above hard to implement and led to several versions of the tool, each with a different functionality. Hence the new version of AUGUR was needed which should have a more general and extensible software architecture and should allow an easier extension of the tool with more functionality concerning analysis and visualization methods.

Another new feature of the tool which was almost impossible to implement in the AUGUR 1, was the possibility to work with attributed graphs, i.e., graphs with (integer and string) attributes assigned to nodes and edges and the extension of existing analysis techniques accordingly. It was also rather hard to plan extensions of input and output, for instance with an interface to AGG [104].

All this led us to the idea of the creation of a completely new tool AUGUR 2 with a corresponding functionality and an easily extendable architecture.

6.2 Software Design of Augur 2

In this section we present the main ideas behind the new implementation, which lead to an open and flexible new verification tool.

The central part of the software design is the concept of *algorithms*, which are implemented as classes. Each program module working with the common data structures should be realized as an algorithm. New algorithms can be added during the whole life time of the system. As examples of algorithms we mention here different operations on Petri graphs (firing of transitions, building the coverability graph, searching for matches of left-hand sides, performing folding/unfolding steps, etc.) and input/output operations (readers and writers from/to different data formats).

All algorithms work with the same data structures. This makes it possible to use some algorithms as sub-operations inside other algorithms and to assemble new algorithms out of existing ones. A *scenario* is a special algorithm which uses as input and output only the external data sources, for instance XML files (Fig. 6.1). Scenarios are at the top level of the system and will usually call other algorithms. A typical example for a scenario is the approximated unfolding algorithm which reads a graph transformation system and outputs a Petri graph.

All algorithms and scenarios are managed in a central database system (see Fig. 6.1). The database consists of several tables, the most important being the *algorithm table*, which is shown in Table 6.1.

Calling Algorithm	Label	Algorithm To Call	History Path
A1	a	A2	P1
A1	a	A3	P2
A1	b	A4	\emptyset
A2	a	A5	P

Table 6.1: Example of the algorithm table in the database system.

The first column of the table represents the name of the algorithm which calls another algorithm as a sub-operation. Then, the second column is the label of the place where the sub-operation is being called. Labels are used by algorithms in order to indicate which kind of other algorithms they intend to call. Then, the information in the

³http://www.ti.inf.uni-due.de/research/augur_1/refinement.pdf

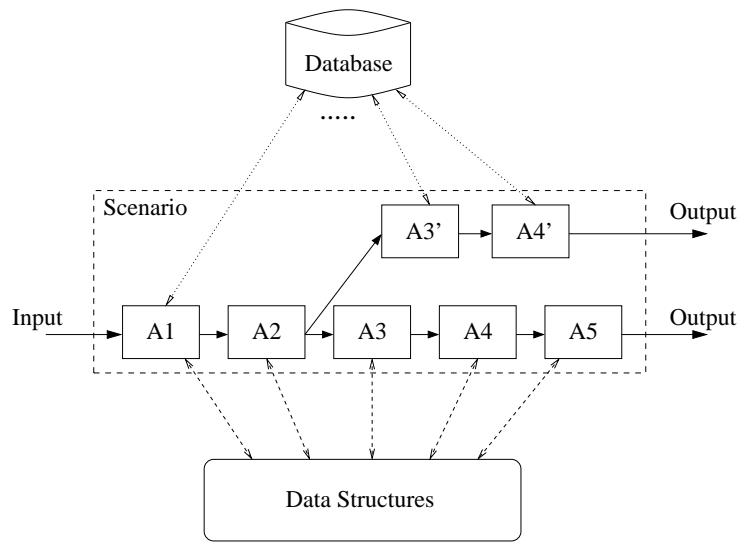


Figure 6.1: Schematic depiction of a scenario.

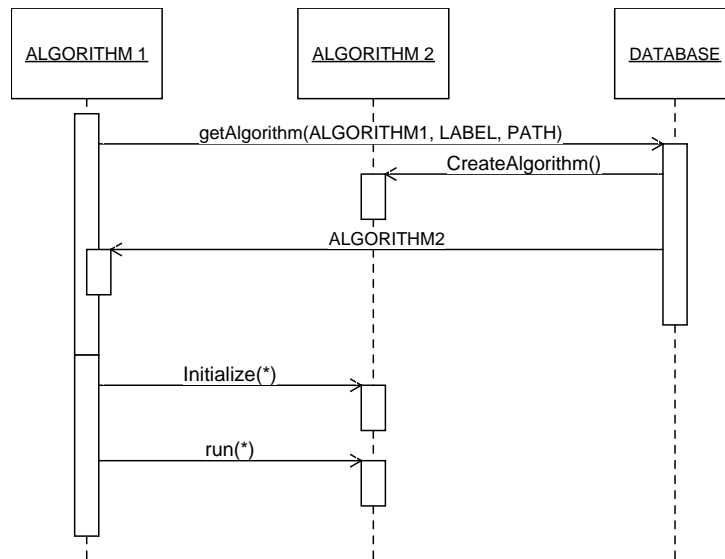


Figure 6.2: Message sequence chart depicting the protocol algorithms have to follow.

database determines which of the several available algorithms for this task is chosen. This information, i.e., the name of the algorithm is given in the third column. Finally, the fourth column is a regular expression representing the dynamic history path or the call stack of the algorithm in the first column. It means that depending on this history different algorithms can be called from the same place in the code. This table makes it easy to exchange a sub-operation by another sub-operation performing basically the same function in a different way (optimized for the concrete situation).

For instance in Fig. 6.1 one can imagine that either algorithm A3 or A3' is called, depending on the current content of the database.

A typical example is the match finder algorithm, which searches for the matches of left-hand sides in a (large) hypergraph. This operation is one of the critical parts in the calculation of the approximating unfolding and there are different ways to implement this operation [17, 110]. Another example is coverability checking for Petri nets, for which we currently use the following different algorithms: computation of coverability graphs [85], backward coverability algorithm [1, 106], approximating reachability computation based on linear equations [111], heuristic search plans [35, 36, 51].

The current layout of the tool makes it easy to replace an old inefficient version of an algorithm by a more efficient one and to use different versions of an algorithm in different situations.

Algorithms calling other algorithms have to follow a certain protocol which is shown as a message sequence chart in Fig. 6.2. Algorithm ALGORITHM1 makes a request to the database in order to look up which algorithm should be called in the code at the place labeled LABEL with the history path PATH. ALGORITHM2 will be created by the database and a reference returned to ALGORITHM1. Then ALGORITHM1 initializes and starts the obtained algorithm. In the following ALGORITHM2 can also ask the database for further algorithms.

Besides the algorithm table there is other information needed to manage the behavior of algorithms. For example, there is a table describing the reusability of algorithms, i.e., which says whether a new object should be created when a new algorithm is requested or if a previously created object can be reused. There also exists a table describing global parameters of the verification procedure. Apart from the protocol mentioned above, there are also some other protocols governing the communication between algorithms. For example, algorithms can notify each other about changes in the data structures (validation protocol).

More detailed description of the software design of AUGUR 2 can be found in the design documentation⁴.

6.3 System Architecture of Augur 2

After describing the general ideas behind AUGUR 2, we will now describe the architecture behind this tool (see Fig. 6.3), which is actually an instantiation of the software design described in the previous section.

⁴http://www.ti.inf.uni-due.de/research/augur_2/design.pdf

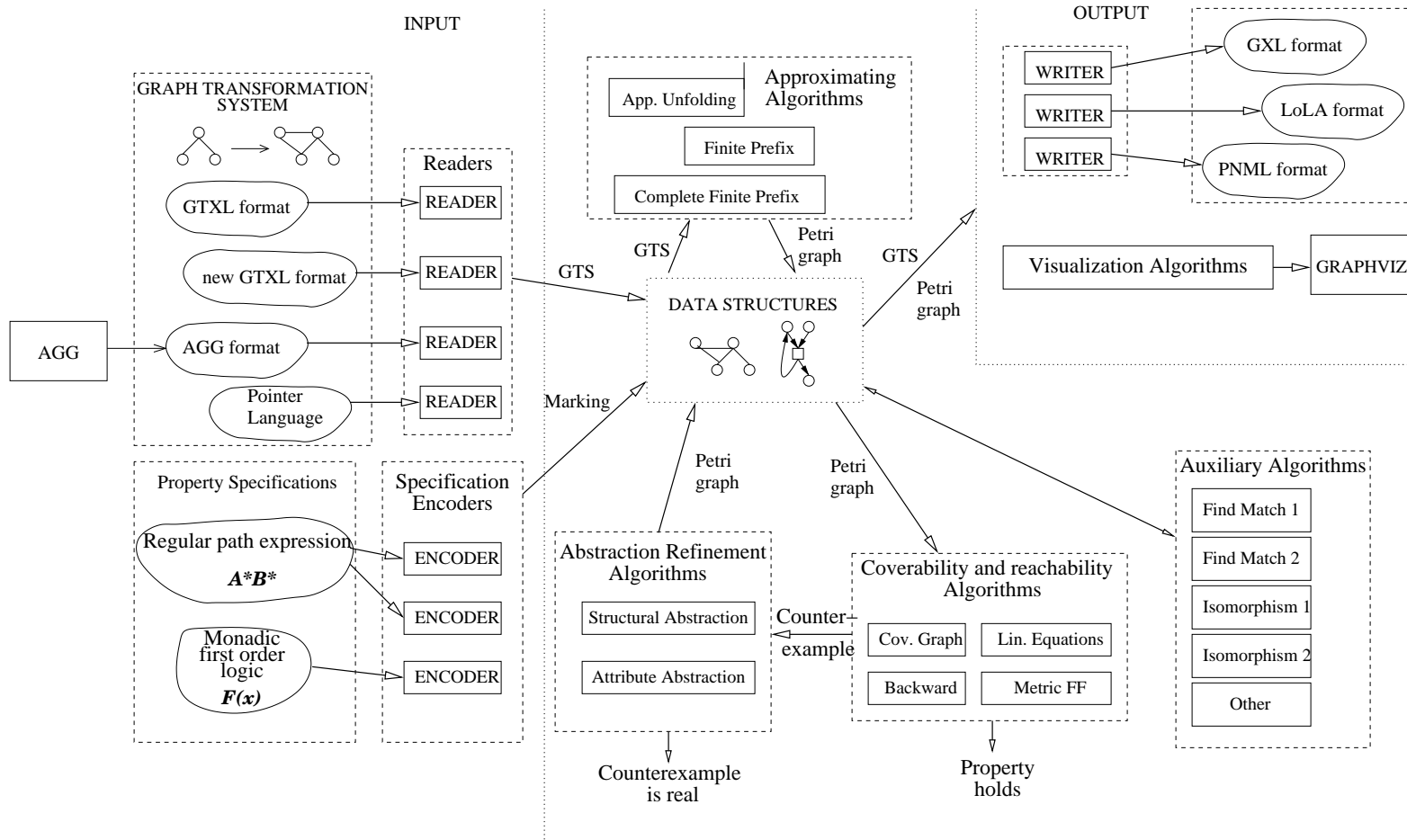


Figure 6.3: Schematic representation of AUGUR 2.

We will explain this figure in roughly chronological order, starting with the input (a graph transformation system and a specification of the property to be verified) and ending with the final output, which says whether the property holds.

The system starts by reading the graph transformation system from an external source. We consider the following possibilities:

- Read the graph transformation system from a file in GTXL-format. GTXL is the XML-based Graph Transformation Exchange Language. The format definition is only available as a draft version at the time of writing⁵. In Appendix B a brief description of the GTXL format based on a small commented example can be found.
- Read the graph transformation system from a file in new GTXL-format. This is a new version of the GTXL-format which is described in [72]. In Appendix C there is a small commented example in the new GTXL format.
- Use AGG as an input source in order to draw graph transformation systems. AGG is a visual tool supporting simulation and some forms of analysis of AGTSs [104]. We use AGG as a visual environment for creation of GTSs and AGTSs. The composed system can be written in a XML-based file format of AGG and can then be read using the special reader-algorithms in AUGUR 2. Detailed information on using AGG as an input tool for AUGUR 2 can be found in the program documentation.
- Write a program in a simple pointer-manipulating language (SPL), which is then converted automatically to a graph transformation system. The corresponding reader of AUGUR 2 was implemented in a diploma thesis [105] where also the formal specification and examples of SPL can be found. In Appendix E we give a small commented example of an SPL program.

After reading the GTS from an input source and converting it to the internal data structures it can be visualized using the Graphviz tool⁶.

The obtained GTS can be abstracted using one of the following algorithms:

- The approximated unfolding algorithm AUNFOLD (see Section 3.2).
- Calculation of a finite prefix of the unfolding with a given depth. The finite prefix is an unfolding of a GTS without folding steps. It is usually constructed until some predefined depth and can then be seen as an under-approximation of a given GTS [8]. The optimized calculation of a finite prefix was considered in [9]. The corresponding algorithm in AUGUR 2 was implemented in a diploma thesis [16].
- Calculation of a finite complete prefix of the unfolding. The technique is based on the unfolding procedure and it generalises McMillans complete prefix approach, originally developed for Petri nets, to graph transformation systems⁷ [8]. The approximation obtained in the case of finite-state GTS represents all reachable hypergraphs. The corresponding algorithm in AUGUR was implemented in a diploma thesis [16].

⁵<http://tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html>

⁶<http://www.research.att.com/sw/tools/graphviz>

⁷The finite complete prefix approach of a Petri net unfolding is a technique introduced by McMillan and later improved by other researchers. The technique is based on so-called cut-off-criteria via which one obtains a finite unfolding having sufficient information about the behaviour of the analyzed Petri net.

One of the critical parts in the calculation of the unfolding is searching for a match of the given hypergraph in some host hypergraph (which is usually larger). A similar problem is checking of hypergraph isomorphism. For these purposes the following algorithms are implemented in AUGUR 2:

- Naive backtracking sub-graph matching algorithm (check all possible combinations of edges).
- Match algorithm using a search-plan heuristic [17, 110]. Although the subgraph matching problem is NP-complete, still in practice it should be performed in reasonable time if possible. For this purpose several heuristics to sub-graph matching were developed. The approach implemented as an algorithm in AUGUR 2 in a diploma thesis [116] uses cost models for the possible matching strategies which takes the structures of both matched graph and host graph into account.
- Naive backtracking hypergraph isomorphism check.
- Isomorphism check using hypergraph certificates. In [90] it is shown how one can use element-based graph certificates in order to help recognising non-isomorphic graphs. Mappings are constructed that assign to all elements of a given graph a number that is invariant under isomorphism. This means that any isomorphism between graphs is sure to preserve this number. This is used as a heuristic to decide whether two graphs are isomorphic or not. The adaptation of this techniques to hypergraphs is implemented in AUGUR 2.

Note that sub-graph isomorphism algorithms are implemented in two different versions. In the first version we only search for the first appearance of a sub-graph in the host graph. In the second version we search for all possible matches.

The obtained unfolding can be written in the external data formats (with the help of the writer-algorithms):

- Graph Exchange Language (GXL) data format [114, 53] (see also <http://www.gupro.de/GXL/>). GXL is a standard XML-based exchange format for graphs which offers an adaptable and flexible means to support interoperability between graph-based tools. In Appendix D there is a small Petri graph specified as a commented GXL.
- LoLA format [100]. LoLA (a Low Level Petri Net Analyzer) is a software tool which has been implemented at the Humboldt-University of Berlin for the validation of Petri net reachability graphs. LoLA can analyze reachability of a given state or state predicate, boundedness of the net or a place, deadlocks, dead transitions, reversibility, and existence of home states. LoLA can also verify formulas of a branching time logic. AUGUR 2 can output a Petri net component of the obtained Petri graph in a LoLA input format.
- Petri graph visualized using the Graphviz tool⁸.

Graphviz is a collection of software tools which provides graph visualization for rather general abstract graph structures. In our case we visualize hypergraphs and Petri nets via two different algorithms (see the Graphviz documentation for more details). The tool is distributed under an open source license and is called from AUGUR 2 as an external application.

⁸<http://www.graphviz.org/>

- PDDL (Planning Domain Definition Language) data format [34]. PDDL is an attempt to standardize planning domain and problem description languages. AUGUR 2 can output a Petri net component of the obtained Petri graph in a PDDL input format which can be later analysed by any planner supporting PDDL standard. We use Metric FF planner [51] in order to solve the coverability problem (see below).

Apart from the graph transformation system, we require the property which has to be verified as additional input. The property is described in some specification language which has to be translated into properties on Petri net markings, since the analysis has to be done directly on the Petri net structure underlying the Petri graph. This conversion is done by the so-called encoders. In AUGUR 2 we consider two possible specification languages

- Regular expression with the set of hyperedge labels as the alphabet. This regular expression describes forbidden paths which should not occur in any reachable graph [54]. The encoder-algorithm has been implemented in AUGUR 2 in a student project [86].
- Monadic first-order logic on hypergraphs [13]. The logic is expressive enough to characterise typical graph properties, and can be effectively verified. The encoder-algorithm has been implemented in AUGUR 2 in the frame of a diploma work [111].

Note that these specifications do not work well with abstraction refinement (see Section 5.1 for more details). In order to apply the abstraction refinement procedure we usually add a special error rule which have a single error edge with arity 0 as a right-hand side.

For regular expressions we have implemented two different encoders. The first is based on the paths in hypergraphs and produces the markings of Petri nets [54] and the second uses the cross-product of the automaton corresponding to the regular expression and the hypergraph and produces semilinear sets on Petri nets (which describe sets of markings) [65]. For first-order logic on hypergraphs an encoding of such graph formulae into quantifier-free formulae over Petri net markings is implemented.

The coverability of the obtained markings can then be checked using various algorithms described below:

- Construction of a coverability graph (see Section 2.1).
- Backward coverability algorithm (see Section 2.4).
- Approximating reachability computation based on linear equations [111].
- Metric FF planner [35, 36, 51]. In general a planner has a description of the world in some formal language, a description of agent's goal (also in some formal language) and a description of the possible actions that can be performed. The planner's output is usually a sequence of actions achieving the goal from the initial state in the frame of the described world. We use the PDDL language [34] as a formal language and the Metric FF planner as a solver of the coverability problem. In AUGUR 2 the obtained Petri net component is output in PDDL format and the Metric FF tool is called as an external application in order to check if the corresponding marking (the goal in the Metric FF tool) can be covered by applying some sequence of transitions (sequence of actions in the Metric FF tool).

Let us note here that in the current implementation of AUGUR 2 only the first two (of the last four) algorithms can analyze the attributed Petri nets. The support of attributes in the approximated reachability algorithm and in the heuristic search plans is a topic of future work.

If the property does not hold, a counterexample for the net is generated. In the case of spurious counterexamples one of the refinement algorithms is used to obtain a more exact approximation. This procedure can be iterated. For the refinement of the obtained approximated unfolding there are two possibilities:

- Structural refinement of the obtained Petri graph (see Section 5.1).
- Refinement of the attribute abstraction (see Section 5.2).

Whenever a non-spurious counterexample is found, we have detected an error in the GTS, i.e., the property to be verified does not hold.

The whole verification procedure implemented in AUGUR 2 is schematically depicted in Fig. 6.4. We have a GTS and a property we want to verify as an input of the system. First of all we construct a Petri graph which is an over-approximation of the given GTS. In the analysis block we first calculate the Petri net marking corresponding to the property to verify (which is obtained from a regular expression on the hypergraph structure of the Petri graph or from a first order logic formula). The marking is then analysed by some coverability algorithm. If the marking is not coverable, then we terminate with “VERIFIED”. This means that the corresponding sub-graph (described by the regular expression and a corresponding marking) cannot be reached during the reduction of the GTS. Otherwise we have two possibilities. The obtained trace to the coverable marking (counter-example) can be real or spurious (i.e., reproducible only in the over-approximation and not in the original GTS). In the case of a real counterexample we terminate with “PROPERTY FALSE”. Otherwise we start a counterexample-guided abstraction refinement procedure and obtain a refined Petri graph. The refinement procedure can be iterated a predefined number of times. If we still do not have a verification result, then we terminate with “UNKNOWN”. For each operation a timeout is set such that when it is reached, the verification process stops with “TIMEOUT”. We say that the verification problem for GTS is solved if the property is verified or we have found a (non-spurious) counter-example.

6.4 Functionality and Usage

The tool AUGUR 2 is implemented in C++ and can be called as a command-line application or using the graphical user interface (GUI) written in Java. The implementation was done in Linux. The user should have Latex, Graphviz⁹, the library LIBXML 2¹⁰, the library LP Solve¹¹ and optionally the compiled tool Metric FF¹² pre-installed on the computer. For the graphical user interface Java with version number at least 1.5 is needed. The complete description of usage can be found in the program documentation¹³. Here we describe only the basic functionality and the main options of the AUGUR 2.

First we describe the database settings, which should be made before the tool is used. The database is implemented as a single XML-file and consists of three parts. In the first part there is a description of the global parameters of the verification procedure. Below

⁹<http://www.graphviz.org/>

¹⁰<http://xmlsoft.org/>

¹¹<http://lpsolve.sourceforge.net/>

¹²<http://members.deri.at/~joergh/metric-ff.html>

¹³http://www.ti.inf.uni-due.de/research/tools/augur/doc_augur_2.pdf

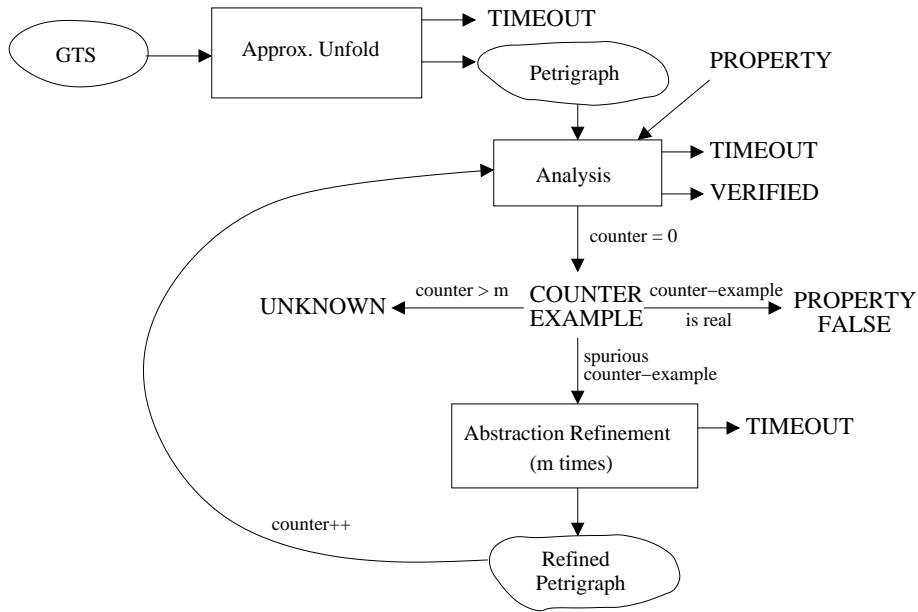


Figure 6.4: Verification technique

we give an example for the format of global variables. In the example, two parameters are boolean and the third is of the type integer. The complete list of global parameters can be found in the program documentation.

```

<Globals>
  <parameter_1 val="true"/>
  <parameter_2 val="false"/>
  <parameter_3 val="10"/>
</Globals>

```

The second part describes algorithms and their interaction in the format as in the example below. Here for the algorithm *alg_1* which is not reusable (see below) at the label *l_1* we have following possibilities: if in the stack of the current instance of *alg_1* the algorithms *alg_3* and *alg_4* were called at the labels *l_3* and *l_4* respectively then the algorithm *alg_5* will be called. Otherwise we call the default algorithm *alg_2*.

```

<algorithm name="alg_1" reusable="false">
  <label name="l_1">
    <default algorithm="alg_2"/>
    <info>description</info>
    <expert status="true">
      <history>
        <happen algorithm="alg_3" label="l_3"/>
        <happen algorithm="alg_4" label="l_4"/>
        <call algorithm="alg_5"/>
      </history>
    </expert>
  </label>
</algorithm>

```

The third part of the database file is a description of scenarios which are algorithms having only external files as input and output sources. Scenarios are high-level algorithms describing the current task of the tool. In the database we specify which algorithm will be used in each scenario. For example in the scenario *unfold* (see below) it can be either *approximating unfolding* or *finite prefix* or *finite complete prefix*.

```
<algorithm name="main" reusable="false">
  <label name="scenario_1">
    <default algorithm="alg_1"/>
    <info>description</info>
    <expert status="false"/>
  </label>
  <label name="scenario_2">
    <default algorithm="alg_2"/>
    <info>description</info>
    <expert status="false"/>
  </label>
</algorithm>
```

For each algorithm we describe in the database if the algorithm is reusable or not. Non-reusable algorithms will be recreated each time, when they are called, whereas for reusable algorithms the same instance will be used during the whole session.

The program is called from the shell according to the following format:

```
augur -db=DATABASE -sc=SCENARIO [options] input_files output_files
```

The field DATABASE should contain the path to the XML file. In the field SCENARIO we write an alias of the scenario to be executed. The following scenarios are currently available in AUGUR 2 (for the algorithm aliases see the program documentation):

- **aunfold**: Construction of the unfolding. Possible algorithms are *approximated unfolding*, *finite prefix*, *finite complete prefix*. The input file is a GTS and the output file is a Petri graph.
- **property2marking**: Encoder from a property (which is a regular expression or a first-order logic formula) to a marking. Possible algorithms are *simple path encoder* - *sponge*, *encoder to semilinear set* (the marking is then the first element of the semilinear set), *logic encoder*.
- **cover**: Checking the coverability property for the given marking. Possible algorithms are *coverability graph*, *backward coverability*, *approximated reachability*, *metric FF*.
- **refinement**: Counterexample-based abstraction refinement. Here the kind of refinement (structural or attribute-based refinement) will be detected automatically.
- **refinement_loop**: Full automatization of the verification procedure (see Fig. 6.4).
- **gts_emulator**: Emulates the application of the rules in the given GTS.
- **rules2ps**: Visualizes the given GTS and writes it to a postscript file.
- **hg2ps**: Visualizes the hypergraph component of the Petri graph.
- **pn2ps**: Visualizes the Petri net component of the Petri graph.

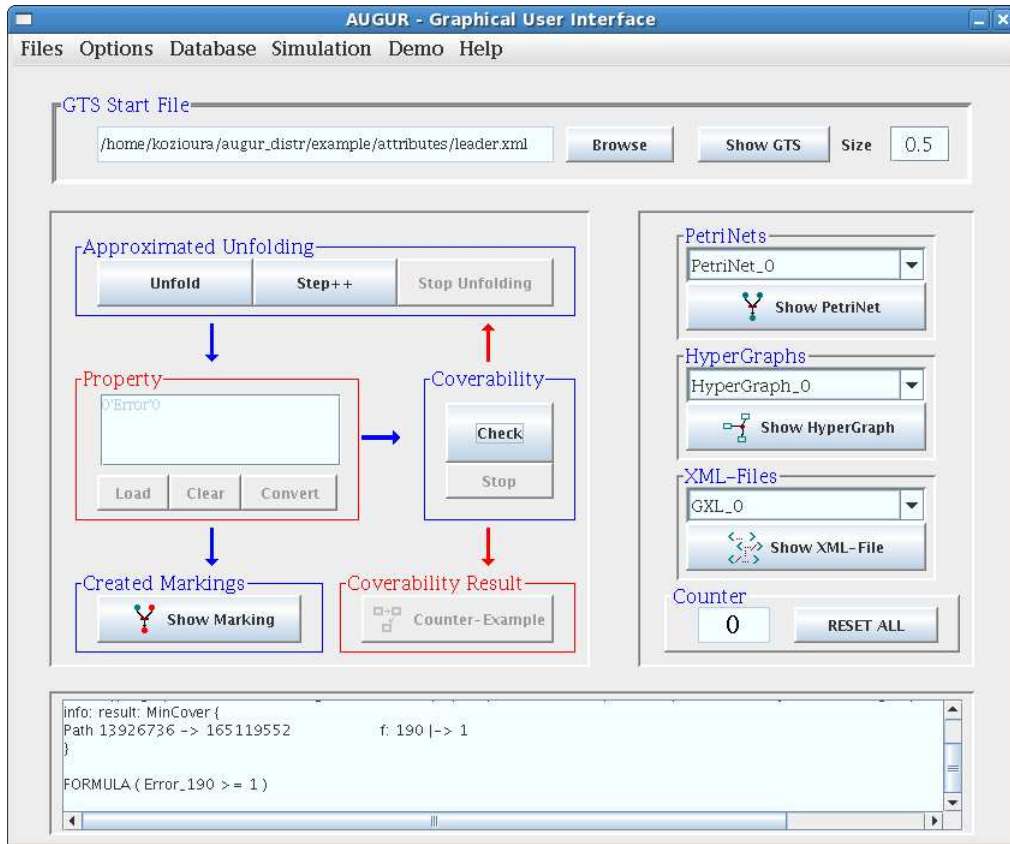


Figure 6.5: Screenshot of the GUI Panel

- **sp1**: Converts an input in the simple pointer language format (SPL) to the GTXL format.
- **test**: This scenario is used for debugging and test purposes.

The complete list of options can be found in the program documentation. We mention here for example the option turning off the coverability check during the unfolding *-nc* and the options for the debug *-d* and quiet *-q* modes.

As it was mentioned above the tool can also be used from the graphical user interface (GUI) (Fig. 6.5). We give here only a brief sketch of the its usage. For more information see the GUI documentation¹⁴.

The following options should be set in the options panel of the GUI: postscript viewer (default gv), web-browser (default firefox), editor for XML files (default emacs), editor for text files (default emacs), path to the current installation of AUGUR, path to the working directory.

The GUI has two modes for unfolding and analysing graph transformation systems and one mode emulating the behaviour of graph transformation systems:

1. The standard mode which is visible in the start window (Fig. 6.5). In this mode each single step of the verification can be controlled.

¹⁴<http://www.ti.inf.uni-due.de/research/augur/gui.pdf>

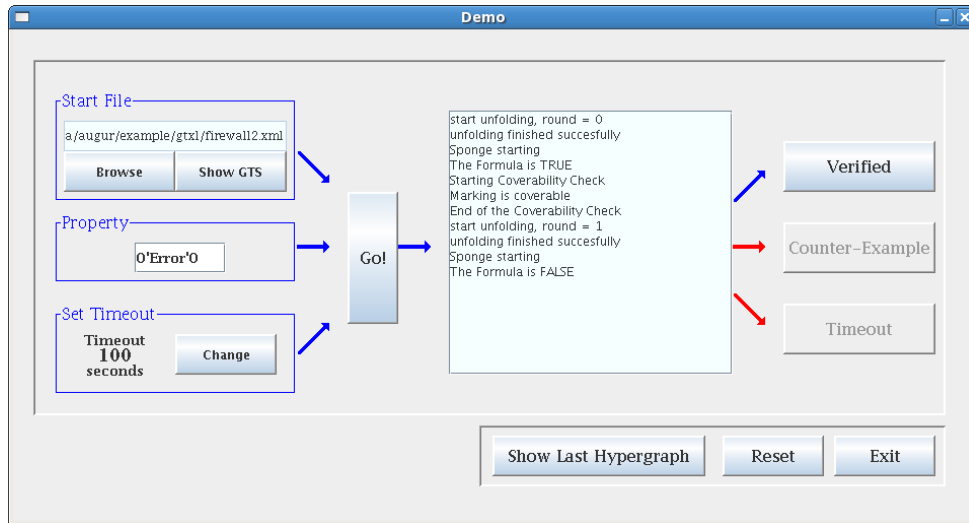


Figure 6.6: Demo Panel of the GUI

2. The demo mode (depicted in Fig. 6.6) starts the complete verification process (including the abstraction refinement loop) by pressing a single button.
3. The simulation mode allows one to apply chosen rules to the current hypergraph.

In the standard mode one starts by choosing a GTS via **GTS Start File**. This GTS can be visualized with the **Show GTS** button. If the initial graph or the rules get too large, the size of the graphs can be modified via the parameter **Size** (default: 0.4). Now the unfolding procedure can be started by pressing **Unfold**. With the button **Step++** it is also possible to do unfolding step-by-step. The unfolding process can be stopped at any time by hitting **Stop Unfolding**. Then, after the unfolding procedure has finished one can specify the property to be verified as a regular expression or a logic formula in the text box **Property**. It might be for example the regular expression **Sprv C* Spub** or **Error** (see the program documentation for more details). Alternatively, the property can be loaded from a file. By pressing **Convert** the regular expression will be converted into a marking of a Petri net which can then be inspected via **Show Marking**. As a next step, the coverability of this marking can be checked using the **Check** button. If the coverability check takes too long, it can be always aborted using the corresponding **Stop** button. If the marking is coverable (this will be shown in the output window) one can see the computed trace to the marking by pressing **Counter-Example**. If the marking is coverable (and hence the non-reachability of the regular expression above could not be verified) one can start the next verification cycle (with abstraction refinement) by pressing again **Unfold**. **Counter** shows the number of the current verification step.

On the right-hand side of the panel there are visualization buttons **Show PetriNet** and **Show Hypergraph**. With the help of the pull-down menus one can see the Petri nets and hypergraphs obtained after each verification step. One can also inspect the corresponding XML-files by pressing **Show XML-file**. With the **Reset** all button or by choosing a new GTS via **Browse** the verification procedure can be restarted.

In order to enter the Demo mode (Fig. 6.6), one should change the view in the GUI by choosing **Show Demo** in the Demo menu. A new window appears where one can choose a start file (containing the GTS), enter a regular expression (**0>Error*0** is the default value) and the timeout (default: 100 seconds). Then the whole verification process can

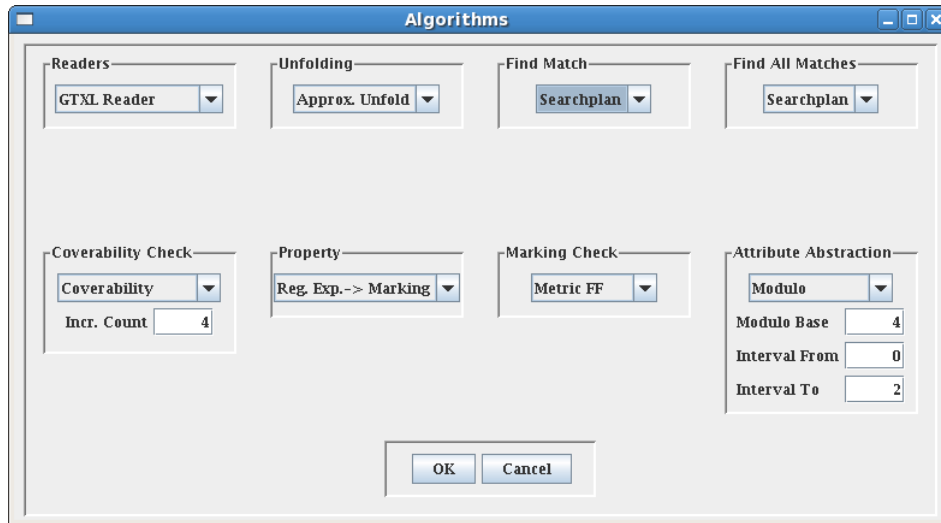


Figure 6.7: Algorithm Dialog of the GUI

be started by simply pressing the **Go!** button and tool will make all necessary verification steps automatically.

The possible answers of the tool in demo mode are:

- Verified - if the property is successfully verified. In this case one can see the hypergraph obtained in the last step.
- Counter-Example - if some real (non-spurious) counter-example has been found.
- Timeout - if the verification task could not be solved in time due to the predefined timeout.

In order to enter the simulation mode choose "Start Simulation..." in the "Simulation" menu. Here one can start with the initial hypergraph of the considered graph transformation system and choose at each step the rule to be applied. The simulation mode works in the standard gnome terminal. Furthermore, by choosing "Visualize Hypergraph" in the "Simulation" menu one can visualize the hypergraph obtained after arbitrary step.

Besides the functionality described above, the GUI also simplifies working with a database. Using the **Database** menu and the dialog panels some important global parameters and algorithms can be easily set (see Fig. 6.7 for the dialog with algorithms).

Chapter 7

Case Studies

In this chapter we discuss some applications of the verification technique presented in this thesis. We consider systems described as GTSs or AGTSs and apply the over-approximating unfolding and abstraction refinement methods.

In Section 7.1 we consider a "Public-Private Server" system described as a (non-attributed) GTS. The system can be seen as an extension of our running example and the key point here is a comparison between the depth-based refinement approach and the counter-example guided abstraction refinement. The last has an advantage both in run-time and in the size of the constructed over-approximations.

The second example (Section 7.2) is called "Firewall" and is also a (non-attributed) GTS which was successfully verified using the techniques described in this thesis. Here the verification can be made only by using the counter-example guided abstraction refinement technique. The approach based on the increasing depth of the constructed unfolding leads to an infinite chain of refined over-approximations and does not allow one to verify the system.

In the next section (Section 7.3) we describe the experimental results obtained by the optimization of the unfolding procedure based on the incremental coverability technique described in Section 2.3. We show that using the incremental approach for the coverability problem of Petri nets it is often possible to speed-up the construction of the over-approximating unfolding.

In Section 7.4 we gather some statistical results of the verification technique of GTSs based on the over-approximating unfolding and abstraction refinement method. In order to obtain statistical results we generate some random graph transformation systems and apply the verification procedure to them.

Section 7.5 represents verification of insertion of elements into red-black trees (a form of balanced search trees) using GTSs. First red-black trees and operations on them are modelled using GTSs. Then we use approximated unfoldings in order to show that insertion preserves the property that there are no two consecutive red nodes in a tree (a requirement for red-black trees).

In Section 7.6 we describe the verification of a leader election protocol in a ring topology. In the protocol we need an integer counter for identifying generated stations and messages. The system is described as an AGTS with an arbitrary large number of stations and messages and integer data type for the IDs of stations.

The last case study described in Section 7.7 is the modelling of the well-known attack on the Needham-Schroeder protocol in the frame of AGTSs. Here we show more possibilities of using attributes for the description of a complex system. Before we demonstrate the attack on the protocol as a real counterexample, we must make two refinement steps in order to eliminate the spurious runs.

All case studies were solved using the verification tool AUGUR described in Chapter 6 under the operating system Linux. The results from Section 7.4 were obtained using the previous version of the tool: AUGUR 1. All other case studies were solved using AUGUR 2. The first five sections (7.1–7.5) have been done on 2×Xeon 2.4 GHz, 2GB RAM and the last two sections (7.6 and 7.7) have been done on 2×Genuine Intel(R) 1.66 GHz with 2GB RAM.

7.1 Public-Private Servers

In this section we describe the verification of the system “Public-Private Servers” (Fig. 7.1), which can be seen as an extension of the running example from Section 3.1. This system consists of public and private servers linked by network connections. Generators produce an unlimited number of public servers and one private server. The servers in turn produce mobile processes (internal processes by the private and external by the public servers). New connections can be created between the servers, where however no connection is allowed from a public to a private server. Processes may cross these connections. Furthermore at some point in time the private server may decide to become a public server.

The properties we want to verify are:

(NC) No connection will ever be created from a public to a private server (only connections going in the other direction and connections between public servers are allowed).

(EP) External processes will never access private servers.

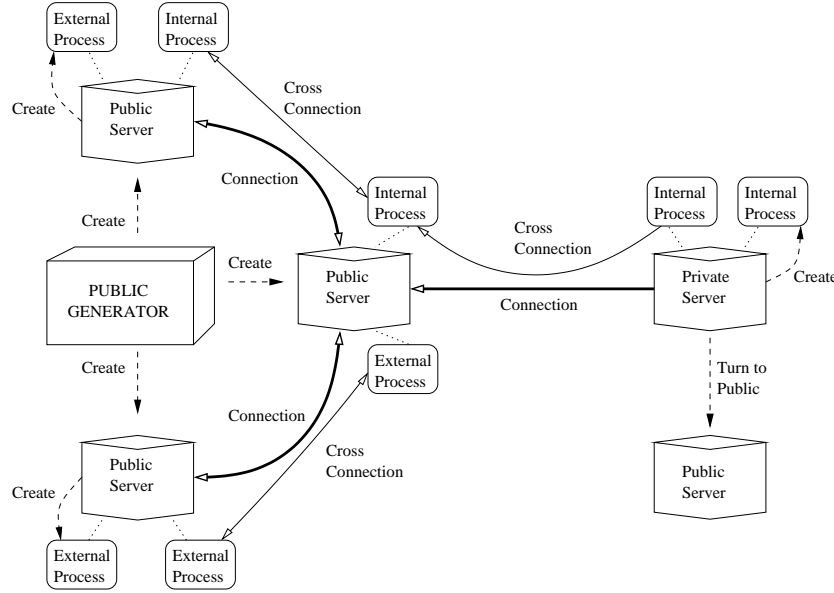


Figure 7.1: Public-Private Servers

We consider two variants of this system by using two different initial graphs: containing a private server and a public generator or containing only two generators (public and private) and a set of rules describing the transformations schematically depicted in Fig. 7.1.

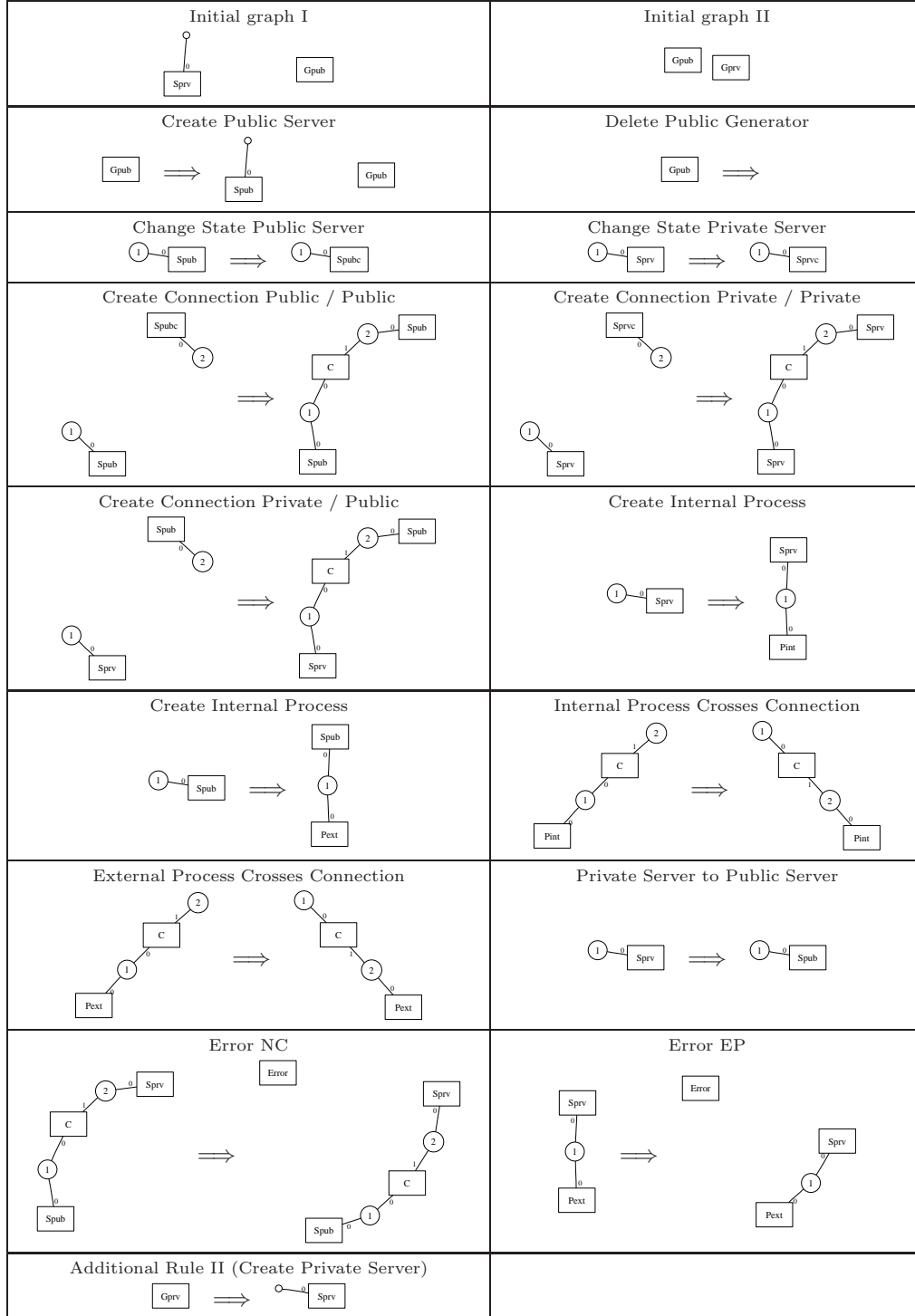


Figure 7.2: Public/Private Servers I and II

The GTSSs modelling the Public-Private Servers can be seen in Fig. 7.2. Note that versions I and II have only two differences. First, the second version has a different initial graph as described before. Furthermore, the second version has an additional rule (“Create Private Server”). All other rules are identical for both versions. The layout of the GTSSs was automatically generated using the Graphviz tool in the way it is done in the AUGUR environment. Here the numbers in nodes denote the values of the mapping α (see Definition 3.1.2) and the numbers on arcs between nodes and edges denote the values of the connection function c_G (see Definition 3.1.1).

The choice of taking an alternative initial graph was made in order to show how slight variations can affect the previous abstraction refinement technique (exact unfolding up to depth k), an effect that disappears by using the counterexample-guided abstraction refinement.

For each property we want to verify, we insert a special “Error” rule. The meaning and the necessity of inserting this rules in order to obtain the refined over-approximation is explained in Section 5.1.

For the verification we use AUGUR 2 and its scenarios as described in Section 6.4. In order to verify the “Public-Private Servers” system, we first construct the approximated unfolding using the scenario `unfold` (construction of the unfolding). Then we call the scenario `property2marking` (encoder from a property to a marking) with the regular expression `0>Error'0` (a single edge of arity 0 labelled with “Error”) in order to obtain a set of markings M with the following meaning: all markings which cover any marking of M are exactly the markings representing “bad” graphs, i.e., graphs containing the edge “Error” and therefore violating the property to be verified. The scenario `cover` (calculation of the coverability property) of AUGUR 2 tells us that indeed some markings contained in M are coverable in the over-approximation, and gives the counterexamples for both considered properties. This means that the obtained 0-depth approximation is too coarse and contains spurious error runs for both properties (**NC**) and (**EP**). Hence, these properties cannot be verified using this approximation.

After checking the obtained counterexamples with the scenario `refinement` (counterexample-guided abstraction refinement) we can see that both counterexamples are spurious. We also obtain a refined over-approximation as described in Section 5.1, which leads to successful verification of both properties. The scenario `cover` tells us that the markings corresponding to the regular expression `0>Error'0` are no more coverable.

The other possibility to refine the over-approximation would be to call in the second iteration `unfold` with a higher level of accuracy (for example with $depth > 0$) instead of the counterexample-guided abstraction refinement. This gives us the same verification result, but the run-time and the size of the obtained approximation are larger.

An advantage of this approach is that the regular expression can be written in a more flexible way. For example for the property (**EP**) we can use directly the regular expression `'Private Server''External Process'`. We can even check the generalization of both properties (**NC**) and (**EP**) by using the following regular expression: `'Private Server''Connection'* 'External Process'` which describes an arbitrary number of connections between public and private server. These properties can also be verified using the over-approximation of depth 1.

The experimental results for the two forms of abstraction refinement that we consider are given in Tables 7.1 and 7.2, which show the size (number of nodes, edges and transitions) of the constructed over-approximation, the run-time and truth values indicating whether the properties under consideration can be verified. The times for successful verification are highlighted by using boldface.

If we compare the results in Tables 7.1 and 7.2, it can be seen that in the case of counterexample-guided abstraction refinement we have an advantage both in the run-time for computing the approximation and in the size of the over-approximations, which

are consequently easier to analyze. The difference is especially pronounced for the second version.

The efficiency of the abstraction refinement approach can be explained by the fact that we forbid to merge only those parts of the unfolding that are responsible for the spurious counterexample. This means that the over-approximation remains rather compact compared to the depth-based (or k -covering) approach, where we are not allowed to merge any items having depth smaller than k .

example	depth	nodes	edges	transitions	time (sec)	verified
Public/private servers I	0	1	9	13	0.05	no
Public/private servers I	1	2	19	34	0.72	yes
Public/private servers II	0	1	10	14	0.05	no
Public/private servers II	1	1	11	16	0.07	no
Public/private servers II	2	3	31	63	7.16	yes

Table 7.1: Verification results (abstraction refinement by forbidding folding steps up to a certain depth k , i.e., by computing k -coverings).

example	nodes	edges	transitions	time (sec)	verified
Public/private servers I	2	16	25	0.67	yes
Public/private servers II	2	17	26	0.68	yes

Table 7.2: Verification results (counterexample-guided abstraction refinement).

7.2 Firewall

The second example describes a firewall system similar to the one introduced in [7]. This system contains an (arbitrarily large) set of processes running behind a firewall (safe processes) and one process in a public area (unsafe process). Any number of safe processes (SP) and connected locations (L) can be generated during run-time. The property to verify is that the unsafe process from the public area does not penetrate the firewall. If this situation is detected, rule “Error” will be applied and an edge labelled *Error* is created.

Fig. 7.3 and Table 7.3 depict the initial graph and the rules of the first version of firewall system. The second version of the firewall example has a different initial graph (Fig. 7.4) and the same rules as the first version. A double-headed arrow in a rule means that the rule can be applied in both directions. Numbers close to the nodes indicate the mapping α . The private and public areas are connected by the firewall (F), and initially there is one unsafe processes (UP) in the public area. Only safe processes will be generated and the firewall can be crossed in one direction only. Our aim is to show that no reachable graph contains the 0-ary edge *Error*.

As in the previous section we take an alternative initial graph in order to show how slight variations affect the different abstraction refinement techniques (exact unfolding up to depth k and the counterexample-guided abstraction refinement).

<p>Create Process</p>	<p>Cross Location</p>
<p>Cross Connection</p>	<p>Cross Firewall</p>
<p>Create Connected Location</p>	<p>Error</p>

Table 7.3: Rules of the firewall system.

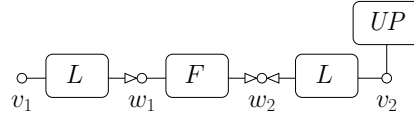


Figure 7.3: Initial graph of the firewall system.

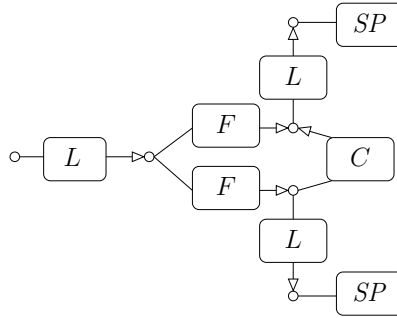


Figure 7.4: Initial graph of the *Firewall II* example

The over-approximation constructed for the first version of the "Firewall" system consists of the hypergraph in Fig. 7.5 and the Petri net in Fig. 7.6. (Ignore the two high-lighted transitions for the moment.) Note that the set of edges of the graph corresponds exactly to the set of places of the net (the correspondence is indicated by giving indices to the labels).

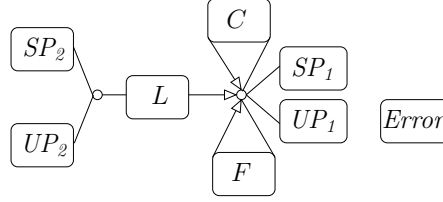


Figure 7.5: Hypergraph component of the approximating Petri graph (firewall example).

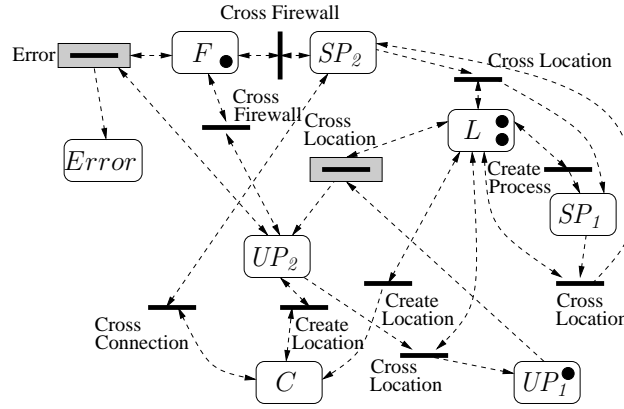


Figure 7.6: Petri net component of the approximating Petri graph (firewall example).

A counterexample in the underlying Petri net, i.e., the run which finally covers the place $Error$, can be found with one of the coverability algorithms from Chapter 2. The first over-approximation (where all folding steps are allowed) contains a spurious error trace ("Cross Location", "Error") highlighted in Fig. 7.6. The counterexample has appeared due to the merging of the nodes before and behind the firewall. In this case the functionality of the firewall is completely broken.

But there also exists a spurious error trace for the 1-covering (where folding of items of depth 0 is forbidden). In fact one can show that the property cannot be verified with any k -covering. The reason for this is that new locations of arbitrary depth are created that are being merged by the approximation, which also holds for locations in front of and behind the firewalls. In this way processes running at locations will—in the approximation—"move around" firewalls without actually crossing them.

In the case of counterexample-guided abstraction refinement newly created locations will be merged with existing locations and this effect does not appear, which means that the property can be verified. The refined hypergraph component of the over-approximation can be seen in Fig. 7.7. As one can see there is no more "Error" edge in the approximating hypergraph. The nodes behind and before the firewall are now separated and the example is successfully verified.

As in Section 7.1 we compare the results in Tables 7.4 and 7.5.

In the "Firewall" example as in the previous section (verification of "Public-Private

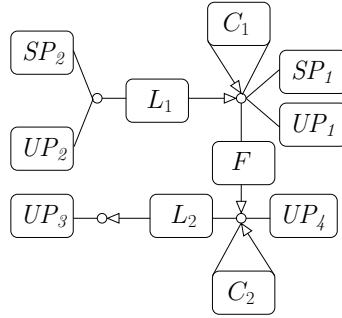


Figure 7.7: Hypergraph component of the refined Petri graph (firewall example).

example	k (depth)	nodes	edges	transitions	time (sec)	verified
Firewall I	0	2	8	13	0.05	no
Firewall I	1	6	25	50	2.4	no
Firewall I	2	10	51	148	138.18	no
Firewall II	0	2	8	13	0.14	no
Firewall II	1	8	39	82	13.7	no
Firewall II	2	14	79	242	858.4	no

Table 7.4: Verification results (abstraction refinement by forbidding folding steps up to a certain depth k , i.e., by computing k -coverings).

example	nodes	edges	transitions	time (sec)	verified
Firewall I	4	11	17	0.16	yes
Firewall II	4	12	18	0.33	yes

Table 7.5: Verification results (counterexample-guided abstraction refinement).

Servers” system), not only the run-time of approximated unfolding is better in the case of counterexample-guided abstraction refinement, but also the Petri net tools checking coverability on the net are substantially faster. For instance, in the case study Firewall II our coverability checker (based on backward reachability) runs for more than one day in the case of the 2-covering.

7.3 Experiments with the Incremental Coverability Approach

Coverability graphs play a twofold role in the verification procedure for GTSs described in this thesis: first, they can be used to analyze the constructed Petri graph in order to infer properties of the original GTS (Section 2.2). Second, they can be used to control the construction of an over-approximating Petri graph (Section 3.2).

The computation of the coverability graphs is a major factor in the run-time of the over-approximating unfolding algorithm and calls for optimizations. Due to the nature of the construction of the over-approximating Petri graph (see Section 3.2) two successive Petri nets are related by a morphism, which in practice changes only relatively small parts of the Petri net.

In Fig. 7.8 we show how to represent a single unfolding and folding steps through the Petri net morphisms. Here $\varphi(s_1) = s_1$ for the unfolding step and $\psi(s_1) = \psi(s_2) = s_1, s_2$, $\psi(s_3) = s_3$ and $\psi(t) = t'$ for the folding step.

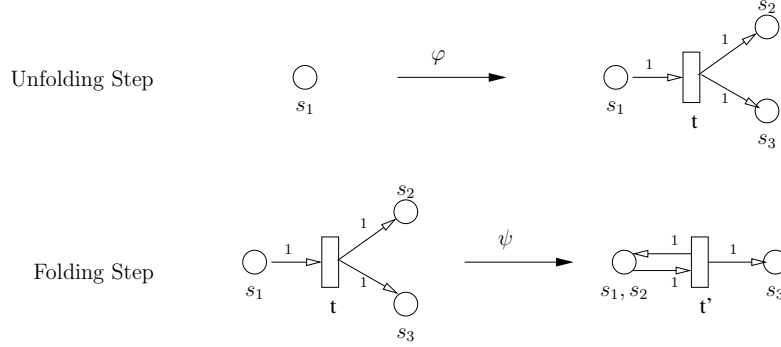


Figure 7.8: Examples of unfolding and folding steps and the corresponding morphisms.

We attempted to speed up the construction of over-approximating Petri graphs by applying the incremental coverability technique introduced in Section 2.3. Below we represent the obtained experimental results. The results of this section are published in [69].

As benchmarks we took, besides Public-Private Server (Section 7.1) and Firewall Example (Section 7.2), several (infinite-state) graph transformation systems (Dining Philosophers Problem [6], Mutual Exclusion Protocol [33] and Red-Black Trees (Section 7.5)).¹

In our experiments we computed Petri graphs by constructing coverability graphs from scratch every time and then by using the incremental technique. Note that for unfolding steps (adding a transition) we have no disadvantage with respect to the incremental technique. However, as it was shown in Section 2.3 in the case of morphisms corresponding to folding steps there is a possible growth of the coverability graph. Our

¹Additional information about the examples can be found at http://www.ti.inf.uni-due.de/research/augur_1/examples/.

experiments with single folding steps have shown that the coverability graph constructed incrementally is in average 5.61% larger than the coverability graph constructed from scratch (which might not be the minimal one). On the other hand the time needed for the construction of the incremental coverability graph is in average only 25% of the standard construction time. For 60% of the nets we have obtained coverability graphs with the same size. In one case the size of the incrementally constructed coverability graph was smaller than the size of a graph constructed in the standard way. The maximal growth of the coverability graph was 39%.

Because of the possible growth of the incrementally constructed coverability graph after several steps the incremental construction usually leads to coverability graphs which become too large. Here we suggest the following approach to this problem: we restart the construction from scratch after a fixed number of N steps. Figure 7.9 shows how the time needed for the construction of the approximated unfolding depends on the parameter N for a specific example (Public/Private Servers II).

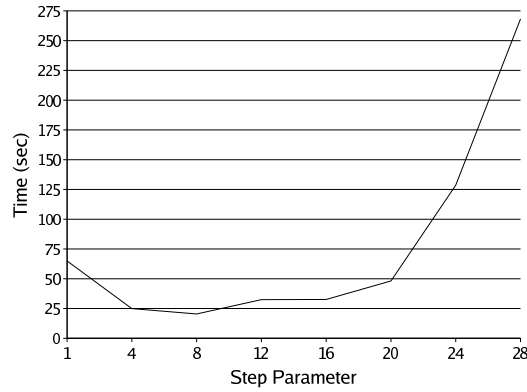


Figure 7.9: Construction time for the approximated unfolding (Public/Private Servers II)

In this way we can fix the parameter N and calculate the coverability graph incrementally for N steps. For our experiments we have chosen $N = 4$ as a default value. Table 7.6 shows the run-times of the algorithm without the incremental technique and with the incremental technique (measured in seconds), indicating a significant decrease in time. The optimal step distance N differs for each of the examples, but in all cases the value $N = 4$ leads to faster computations than $N = 1$.

We have a slightly more detailed look at example Red-Black II [5]: here 55 Petri graphs are created, which means that 55 coverability graphs have to be computed, either from scratch or incrementally. The last Petri net consists of 47 places and 26 transitions and a slightly larger net (52 places, 29 transitions) occurs in an earlier step of the computation. The last net has the largest coverability graph, consisting of 5344 nodes (if computed from scratch).

The first column of this table is the name of GTS, the second column is the calculation time with the standard coverability approach, the third column is the calculation time with incremental coverability and $N = 4$ and the last column is the best time which could be obtained using the incremental approach. The time is measured in seconds.

In Section 2.3 we have shown how to compute coverability graphs in an incremental way when the Petri net is modified as specified by a morphism. A question that naturally arises is whether other modifications to a Petri net besides folding and unfolding steps can be treated in a similar way. We conjecture that all modifications which increase the set of behaviours of the net (such as Petri net morphisms, but also deletion of places) can be treated in an analogous way. In these cases Lemma 2.3.5, which is an essential

GTS	time for $N = 1$	time for $N = 4$	minimum time
Firewall I	1.47	0.94	(N=4) 0.94
Public/Private Servers I	0.46	0.25	(N=9) 0.23
Dining Philosophers	15.63	13.41	(N=7) 12.58
Mutual Exclusion	24.56	23.75	(N=9) 23.31
Red-Black I	39.18	17.20	(N=4) 17.20
Public/Private Servers II	64.98	24.94	(N=8) 20.43
Red-Black II	183.39	96.80	(N=4) 96.80
Public/Private Servers III	312.95	228.19	(N=3) 170.58
Firewall II	483.77	318.54	(N=10) 212.39

Table 7.6: Run-times of the approximated unfolding algorithm (including computation of coverability graphs). Time is measured in seconds.

ingredient of the correctness proofs, holds. If, however, the set of behaviours of a net is decreasing (by deleting transitions or adding extra places to the pre-set of a transition), then incremental techniques seem to be quite problematic. Especially, the construction of a coverability graph inserts ω -places whenever there exists a certain path between two markings. When, however, there are fewer paths than before, some ω -places would have to be reverted back to non- ω -places, which seems to be quite a complex task.

7.4 Random Graph Transformation Systems - Statistical Results

The verification procedure for GTSs based on their over-approximation with Petri graphs (Section 3.2) and the corresponding counterexample-guided abstraction refinement (Section 5.1) remains undecidable in general (because of the Turing-completeness of GTSs). The interesting question is how many GTSs can be verified in practice using the over-approximation of GTSs and standard techniques for analysing Petri nets. This section is an attempt to give an answer to this question. The results of this section are published in [71].

In this section we generate some random GTSs and verify them with the help of AUGUR in order to obtain statistical results. We consider some classes of GTSs identified by a number of parameters. The generated GTSs have hyperedges with arity (number of connected nodes) one or two. Edges are labeled (we consider two labels for each arity). We do not allow two edges having the same labels in the left-hand side of a rule. We also do not delete any nodes. Therefore we describe below only the nodes being added to the right-hand side of the rule.

The following parameters describe the class of generated GTS:

1. Minimal/Maximal number of nodes in the left-hand side of a rule.
2. Minimal/Maximal number of additional nodes in the right-hand side of a rule (see the explanation above).
3. Minimal/Maximal number of edges in the left-hand side of a rule.
4. Minimal/Maximal number of edges in the right-hand side of a rule.
5. Minimal/Maximal number of nodes in the initial graph.
6. Minimal/Maximal number of edges in the initial graph.

7. Minimal/Maximal number of rules.

In this section we consider the following classes of random systems defined by their parameters (minimal and maximal numbers).

system class	nodes LHS	nodes RHS	edges LHS	edges RHS	nodes initial	edges initial	rules
1	1, 2	0, 1	1, 2	1, 2	2, 5	2, 5	3, 5
2	1, 2	0, 2	1, 3	1, 3	2, 5	3, 7	3, 7
3	2, 3	1, 5	3, 7	3, 7	3, 10	3, 10	5, 10

Table 7.7: Classes of random graph transformation systems

The graph parameters are increased in each next class of GTSs and therefore the obtained graphs are getting larger. In each class we generate 100 GTSs. The numbers are relatively small because we tried to keep the sizes of generated GTSs manageable in order to obtain enough statistical material.

In each GTS we insert additionally the special rule “Error”, where the left-hand side is random and the right-hand side consists only of an edge labelled “Error”.

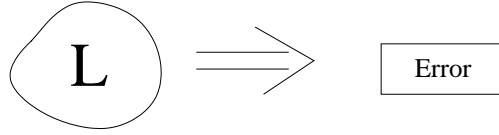


Figure 7.10: Error rule

The property we want to verify is “the Error rule cannot be applied in the generated GTS”, which guarantees that no reachable graph contains L as subgraph. If the rule “Error” can be applied, then the verification algorithm (Fig. 6.4) should give the answer “FALSE” and generate a counterexample. If the rule “Error” cannot be applied, then we should obtain the answer “VERIFIED”. Fig. 7.11 represents an example of a generated GTS from the first class.

We fix 3 iterations for the abstraction refinement procedure and 30 minutes as timeout value. In Table 7.8 average values obtained during the verification of generated systems are represented, namely the number of nodes, edges and transitions in the constructed over-approximations and the verification times (including the timeouts). The verification time is measured in seconds and represents the time of the whole verification procedure.

system class	nodes	edges	transitions	verification time
1	4.21	7.67	4.07	0.01
2	7.47	14.5	10.55	59.87
3	10.01	22.28	25.78	351.53

Table 7.8: Average values of the verified systems.

Diagrams in Fig. 7.12 ((a),(b) and (c), ignore (d) for the moment) describe the distribution of the verification results for the three classes of random systems described above.

An interesting value is also the total number of refinement steps during the verification of one class of GTSs. This value grows rather quickly: 0 steps for the first class of systems,

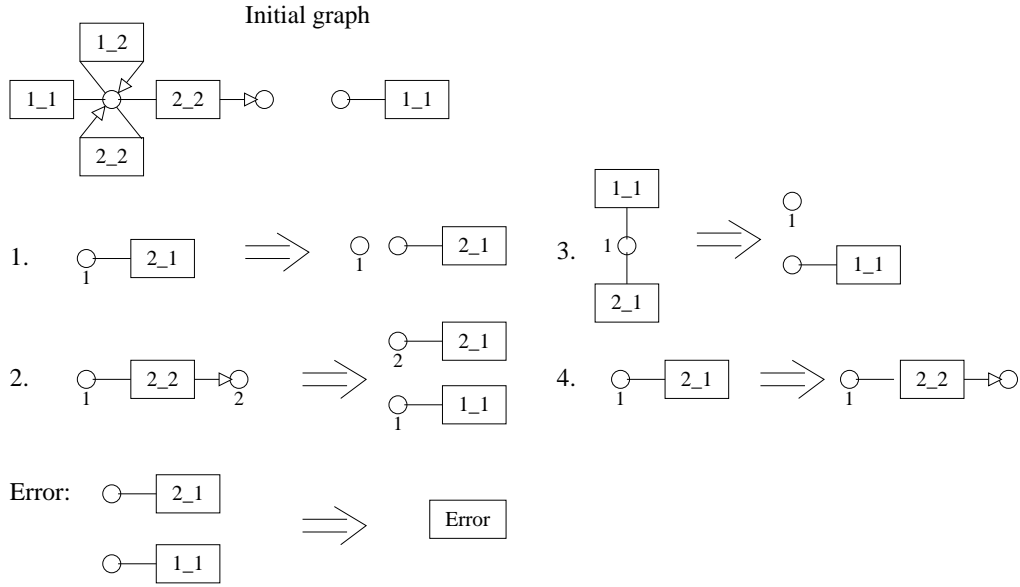


Figure 7.11: Example of a generated GTS (first class of systems)

18 steps for the second class and 83 steps for the third class. But note that the number of refinement steps for each GTS is restricted by 3.

As we can see in Fig. 7.12 we have successfully solved the verification problem for all 100 GTSs in the first class of systems whereas in the third class the number of problems we could not solve is about one third of the number of solved systems. These diagrams give us an idea of possibilities and constraints of the verification approach based on the over-approximation of GTSs with Petri nets. To achieve better verification results we can increase the number of refinement steps and/or the timeout interval. If we start the verification procedure for the same systems belonging to the third class with maximally five refinement steps and with two hours timeout, then we can additionally solve the verification problem for *five* more GTSs, Fig. 7.12(d). The average verification results in this case are represented in Table 7.9. The total number of abstraction refinement steps is 109.

system class	nodes	edges	transitions	verification time
3	11.87	26.57	33.06	1273.74

Table 7.9: Average values for the third class with five refinement steps and two hours timeout

Obviously the systems appearing in real case studies differ from random systems by having a more regular structure, but this experiments give us some (approximative) notion about the possibilities and difficulties of this approach.

The statistical results can be seen as rather positive and hence the verification approach of approximating GTSs by Petri graphs can be seen as a promising approach for the verification of GTSs. It would very interesting to compare these results with related results stemming from other methods, but we are currently not aware of any such results for random systems which have been published.

Some experimental results on the verification of GTSs have been reported in [92]. Note that we are here working in a different setting since we consider potentially infinite

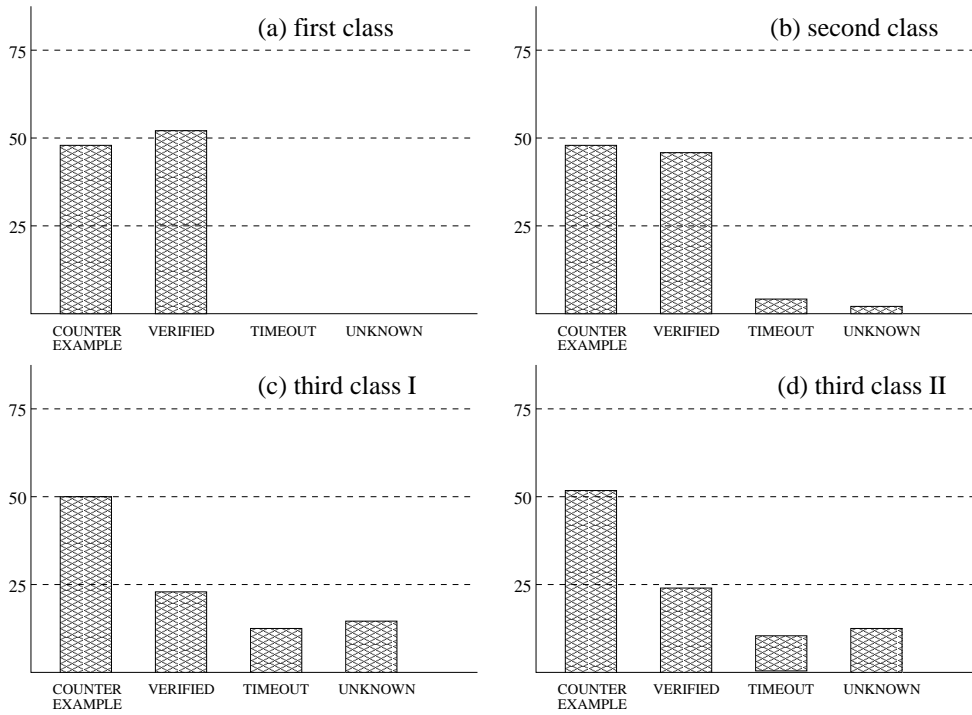


Figure 7.12: Statistic of verification results

state GTSS, whereas [92] considers finite state GTSS.

7.5 Verifying Red-Black Trees

In this section we model the insertion of elements into red-black trees and verify (partial) correctness of the insertion operation. The results of this section are published in [5].

Red-black trees are binary search trees whose nodes are colored either black or red. Only inner nodes can be red, and the following property is satisfied: no red node has a red child. In order to re-establish this property after a new element is inserted, it is necessary to perform some local transformations on the tree (called *rotations*), which have the effect of rebalancing it. Red-black trees are a form of balanced search trees which can be easily implemented (see [82, 27]). They can also be seen as a variant of $(2, 4)$ -trees.

After modeling rotations as graph rewriting rules, we show that the property of red-black trees mentioned above still holds after an insertion. The property is shown by abstracting the obtained GTS by means of the AUGUR 2 tool.

First we introduce red-black trees and their representation as hypergraphs. Then we model insertion into red-black trees using graph rewriting and show how to verify that insertion preserves the structural property of red-black trees mentioned above.

Definition 7.5.1 (Red-black tree) *A red-black tree is a finite binary tree whose inner nodes are associated with keys. Keys are elements of a totally ordered set. A node can either be red or black. A red-black tree satisfies the following conditions:*

- (S) *The tree is sorted, i.e., for every node v the maximal key in the left sub-tree is smaller than the key of v , and the minimal key in the right sub-tree is equal to or larger*

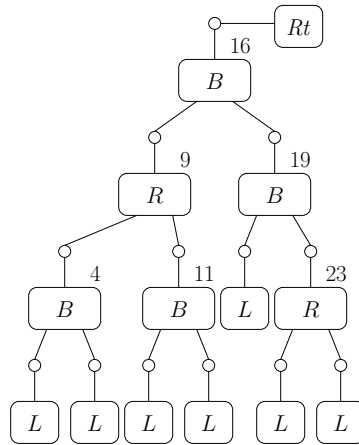


Figure 7.13: An example of a red-black tree.

than the key of v .

- (RL) The root and the leaves are black.
- (D) All leaves have the same black depth, i.e., the number of black nodes on the path from the root is the same for all leaves.
- (R) No path from the root to a leaf contains two consecutive red nodes.

Due to these conditions the longest path from the root to a leaf is at most twice as long as the shortest one. The height of a red-black tree with n inner nodes is therefore $O(\log(n + 1))$, and thus we say that the tree is balanced.

Since we will model insertion into red-black trees by hypergraph rewriting, in the paper we always depict red-black trees as hypergraphs.

A red-black tree is represented as a hypergraph where hyperedges correspond to the nodes of the tree. Inner nodes are represented by hyperedges of arity 3, i.e., they are connected to exactly three vertices, where the parent and the left and right children can be attached. They are labeled by either "R" or "B" depending on whether the node is red or black. Leaves are represented by unary hyperedges labeled "L". Furthermore there is, for technical convenience, a single unary hyperedge labeled "Rt", indicating the root node. Figure 7.13 depicts a red-black tree, where the keys are written next to the hyperedges. Note that, by definition of hypergraph, each hyperedge is connected to an *ordered* sequence of vertices. In our pictures, vertices are always arranged in such a way that the vertex above a hyperedge is its first vertex, whereas the remaining vertices are ordered counter-clockwise.

The insertion of a new node into a red-black tree is described by the hypergraph rewriting rules shown in Fig. 7.14 and Fig. 7.15. For the corresponding pseudo-code, see for instance [82]. An interesting question, that we leave as a topic of future research, is whether and how graph rewriting rules can be synthesized automatically from pseudo-code (some steps in this direction were done in [105] for a simple pointer language).

We remind here that the mapping α maps a node in the left-hand-side to a node of the right-hand side. By numbering the nodes in the left-hand and right-hand sides we map a node in the left-hand-side to the node of the right-hand side with the same number. Furthermore keys are denoted by the letters y, z, u, v .

Rule [add-leaf] describes how a leaf is replaced by a new inner node labeled "M" and two leaves. The label "M" stands for "marker" and denotes a red node during the

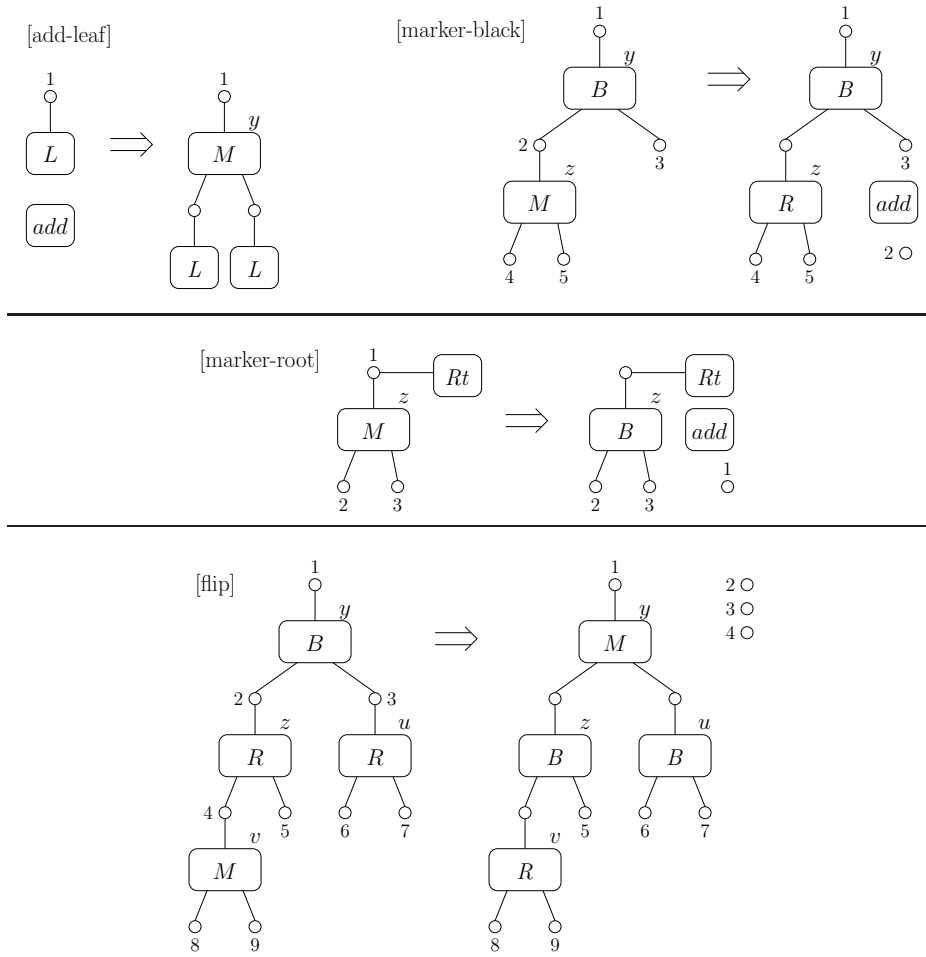


Figure 7.14: Graph rewriting rules (insertion of an element into a red-black tree), part I.

insertion phase. Rule [add-leaf] also consumes a “token”, the 0-ary hyperedge add , that will be generated again when the insertion is completed: this mechanism prevents the concurrent insertion of nodes. We assume that the insertion of the new key y starts from the appropriate leaf, whose position must have been determined by a previous search on the tree. Although this is out of our focus, it is worth observing that this search could be realized by means of graph rewriting rules acting on attributed graphs [78]. The remaining rules describe the local transformations needed to ensure that the tree is converted into a red-black tree.

If the marker has a black parent, it is converted into a red hyperedge and insertion terminates (rule [marker-black], this rule has two symmetric variants). If the marker is the root (rule [marker-root]), it is replaced by a black hyperedge; in this case the black depth of the tree increases by one. If the marker has a red parent, we distinguish several cases (notice that in this case the marker’s grandparent (if any) must be black, because otherwise Condition (R) would be violated):

- If the red parent of the marker has a red sibling, we perform a flip and move the marker upwards (rule [flip], four variants). In this case the algorithm continues.
- If the red parent of the marker has a black sibling, and this sibling is not a leaf, we

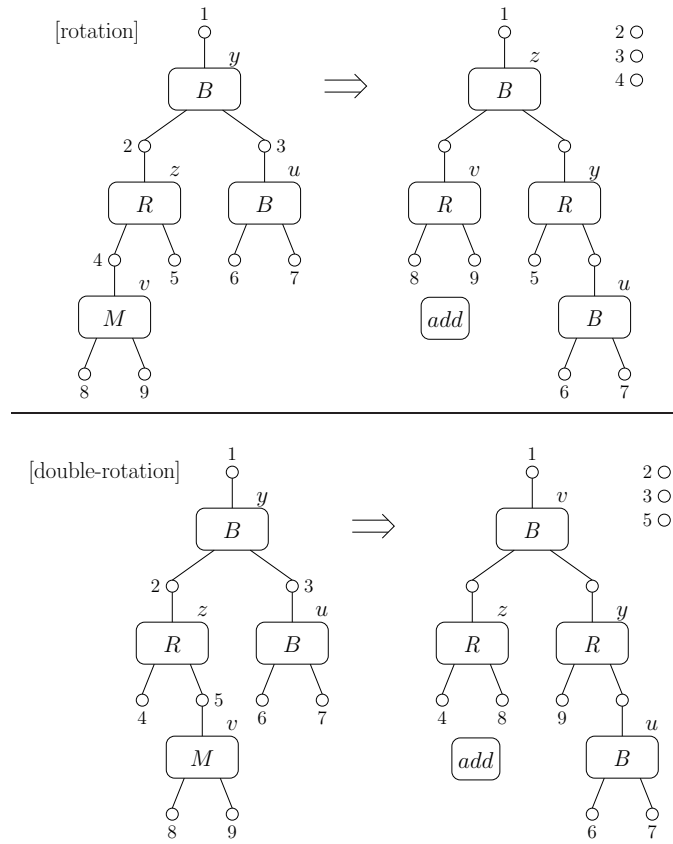


Figure 7.15: Graph rewriting rules (insertion of an element into a red-black tree), part II.

apply either rule [rotation] or rule [double-rotation]. Rule [rotation] is applied if the marker and its red parent are either both left children or both right children. In the two remaining cases rule [double-rotation] is applied. In all cases the algorithm terminates.

- If the red parent of the marker has a black sibling, but this sibling is a leaf, we proceed similarly to the previous case. There are four more rules, obtained from those of Fig. 7.15 by replacing the node with key u by a leaf.

One can see fairly easily that all the transformations expressed by the above rules preserve the sortedness Condition (S) in Definition 7.5.1. Moreover, for any given finite tree, the insertion procedure started by rule [add-leaf] surely terminates, generating again the token *add*. The future work here is a formal verification of these two properties by exploiting the available theory of confluence and termination of graph rewriting systems. In [5] the property that red-black trees remain balanced (Condition (D)) is checked using a suitable type system, which is a simple instance of a general framework [64]. We assume that the preservation of Conditions (S) and (RL) has already been proved, as well as the fact that the result of the insertion procedure is again a tree.

Note that modeling insertion into red-black trees using graph rewriting rules is very natural. Similar diagrams can be found in most text books introducing red-black trees. Usually no marker is used, a red node takes its place instead. However, this would lead to “inconsistent” intermediate states, produced during the insertion procedure, which *do* contain two consecutive red hyperedges, violating Condition (R). We avoid this by using

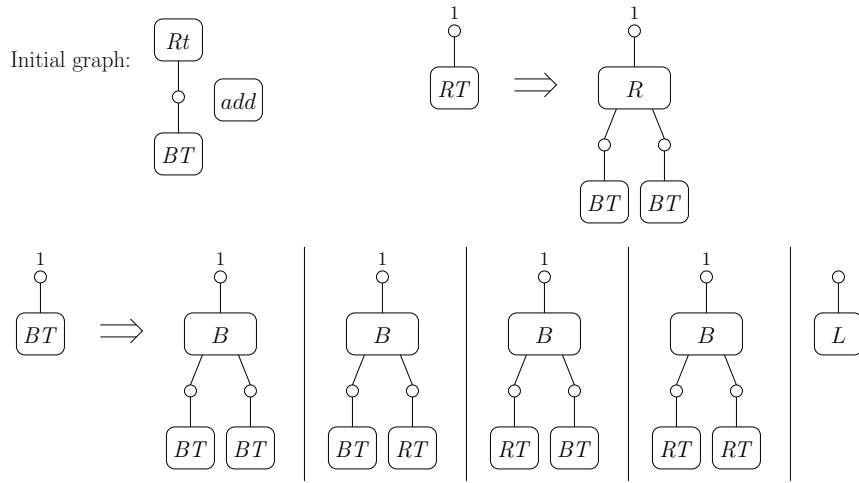


Figure 7.16: A context-free grammar for generating red-black trees.

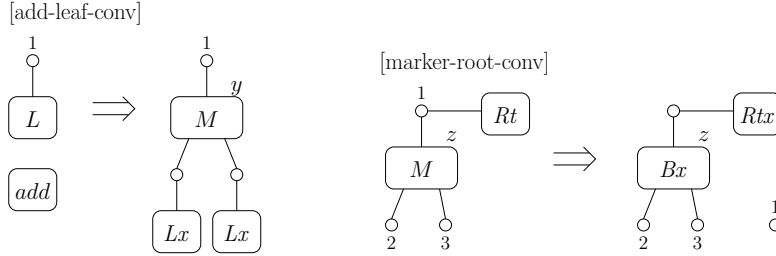


Figure 7.17: Rules of the converted system, part I.

a specific marker, which is furthermore useful for indicating the position in the tree where operations have to be performed.

In order to show with AUGUR 2 that insertion in a red-black tree does not violate Condition (R), we provide as input to the tool a modified version of the rules shown in Figs. 7.14 and 7.15, as well as rules for generating all possible red-black trees. The context-free rules for generating trees are shown in Fig. 7.16, together with the initial graph: they use the non-terminals "BT" and "RT", and generate all finite trees satisfying Conditions (RL) and (R), but possibly not Condition (D) (i.e., they are not balanced). Moreover, the rules modeling insertion are obtained from those of the previous section as described next.

First, since every possible red-black tree is generated by the rules of Fig. 7.16, it is sufficient to show that Condition (R) holds again after a single insertion; thus in the modified rules, the token *add* is never generated again. Second, in order to speed-up the verification, it is convenient to "freeze" the part of the tree traversed during insertion. This is obtained by changing all labels "Rt", "B", "R" and "L" appearing in the right-hand side of rules to labels "Rtx", "Bx", "Rx" and "Lx", respectively, which do not appear in any rule's left-hand side (see Fig. 7.17). This transformation is safe, because the hyperedges with *x*-marked labels do not interfere with the current insertion, and no further insertion is possible by the previous point.

The third modification is necessary because the current implementation of the approximated unfolding suffers from the restriction that a rule cannot have two hyperedges with the same label in the left-hand side (see the program documentation for more details),

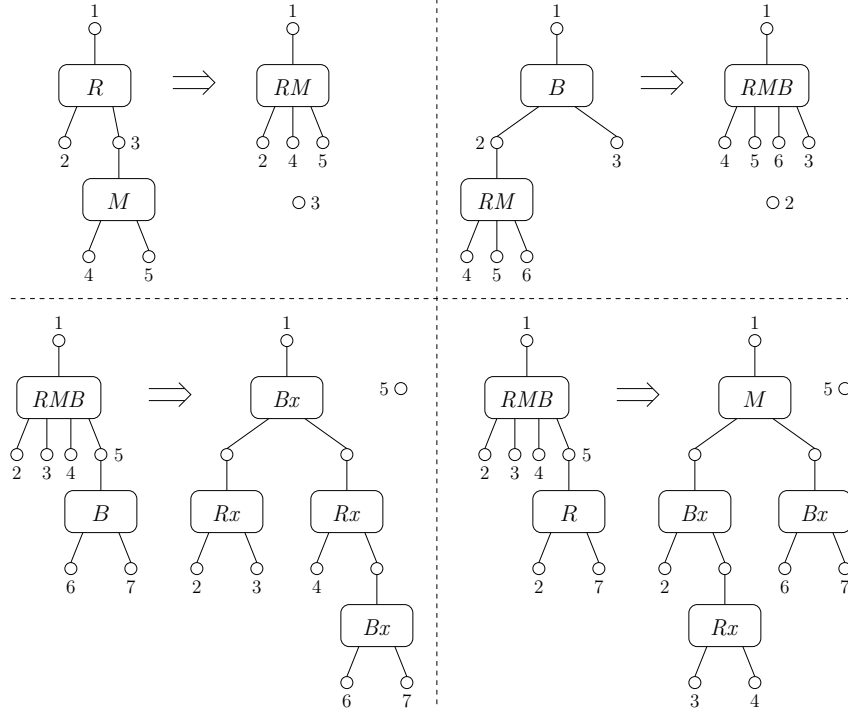


Figure 7.18: Rules of the converted system, part II.

but rules [flip], [rotation] and [double-rotation] do not satisfy this restriction. Therefore the offending rules are converted into an equivalent set of rules which use some new labels and satisfy this restriction. The way the new rules work can be grasped from Fig. 7.18. If the first three rules can be applied in sequence, then we identified an occurrence of the left-hand side of [double-rotation], and therefore the corresponding right-hand side is generated (modified according to the previous two points). If instead after the first two rules the left-hand side of the fourth rule is found, then we generate the right-hand side of a [flip]. It can be shown that the converted rules are equivalent to the original ones, in the sense that if G and G' are graphs containing only labels of the original graph rewriting system, then G can be rewritten to G' in the original system if and only if G can be rewritten to G' in the converted system, possibly in more steps. Furthermore, all hyperedges labeled by a label introduced in the converted system will eventually be deleted.

Applying AUGUR 2 to the graph rewriting system just described and asking for the 0-th approximation we get a Petri graph \mathcal{C}^0 with 125 hyperedges, 72 vertices and 46 transitions, which is too large to be depicted here. In order to show that the property under consideration holds, we want to check that no reachable graph contains a path corresponding to the regular expression $(R + Rx)(R + Rx)$. AUGUR 2 converts this regular expression into a set of markings such that a path of this kind exists in the approximation if and only if the corresponding markings are reachable in \mathcal{C}^0 . However, in this case the set of markings is empty, meaning that the hypergraph underlying the Petri graph does not contain two consecutive red edges. In other words, using only the structural properties of the covering \mathcal{C}^0 (without taking into account its behavior) we can infer the desired property.

7.6 Leader Election Protocol

In this section we describe the modelling and verification of a leader election protocol in a ring architecture [79] with AGTSs. The purpose of the leader election protocol is to elect a unique leader station among the stations in a ring-shaped network (Fig. 7.19). The results of this section are published in [70].

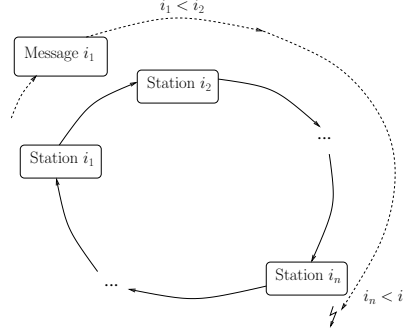


Figure 7.19: Leader Election protocol

The algorithm uses only local communication and does not depend on the ring's size. The leader is chosen based on the unique ids of the stations. Each station sends its id around the ring. Each station compares the obtained ids with its own id and if the incoming id is smaller than its own, then it passes the id through, else the id is discarded. If some station obtains its own id, then it declares itself the leader. This means the leader will be a station having a minimal id.

In our AGTS model on Fig. 7.20 we produce first some number of stations connected in a ring. In this example we allow the number of created stations to be arbitrarily large. The stations are created with the help of the rules “Create First Station” and “Create Station”. We also use additional edge “Counter” in order to have unique ids of the stations. At the moment we consider the generated ring as being ready we apply the rule “Loop Ready”.

A station generates and sends its id with a help of a special edge “Message” and rules “Create Message” and “Send Message”. Message can be sent by a station further if and only if the station's id is larger than the id in the message. This is regulated by the guard expression in the rule “Send Message”.

We also have one special rule which is rather trivial from the protocol's view, but helps us in our verification procedure. We mark one chosen place in the ring (in our case we consider the edge “Counter” as being also such a marker) and the message is considered as passed through the whole ring if it comes back to the station which has sent it and also has visited the marked place in between. This rule helps us with analysing of the over-approximations, where stations can be merged.

The last rule is an “Error” rule, which can be applied if we have chosen the wrong leader, i.e., we have chosen the station having non-minimal id. The property to be verified is: “No Error edge will be created”, i.e., the protocol works correctly.

As in the previous section we take the interval abstraction with the interval $[0, 1]$. After unfolding the AGTS and analysing it using the coverability technique we obtain a counter-example consisting of 7 steps leading to the edge “Error”. Considering this counter-example we see that the approximation is structurally too coarse. After this three iterations of abstraction refinement can be applied: two with structural refinement and one with attribute abstraction to the interval $[0, 2]$. The coverability check of the obtained approximated unfolding with depth one and the attribute interval $[0, 2]$ tells us

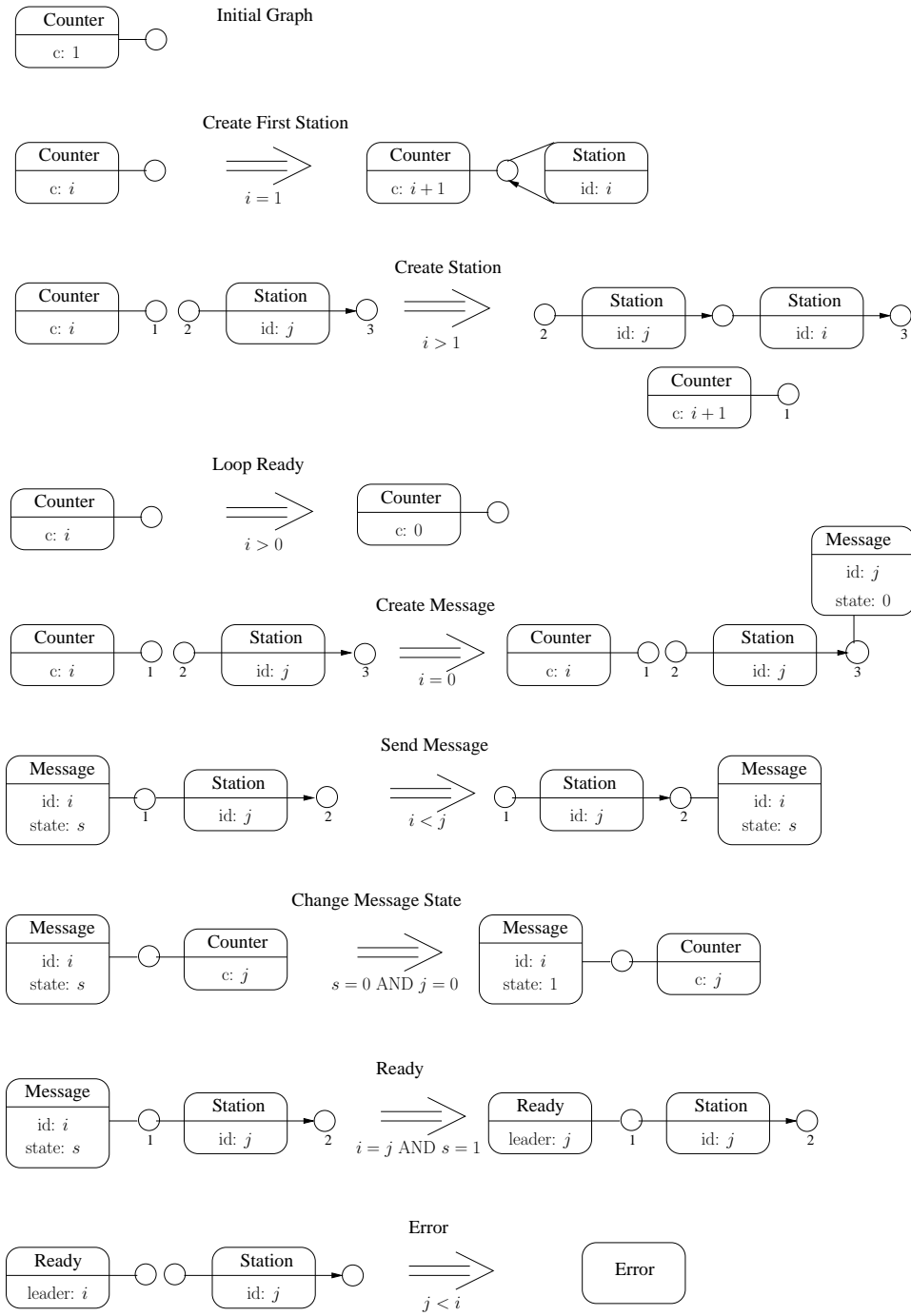


Figure 7.20: AGTS model of Leader Election protocol

that the edge “Error” is no more coverable. This means we have successfully verified the protocol.

The whole verification procedure for the leader election protocol took 48.15 seconds. The resulting Petri graph consists of 2 nodes, 27 edges and 18 transitions.

7.7 Needham-Schroeder Protocol

In this section we model the Needham-Schröder protocol [84] with AGTSs. Our purpose is to rediscover a well-known attack on the protocol found by Gavin Löwe in [75].

The Needham-Schröder protocol is an authentication protocol between two partners consisting of the following three steps (Fig. 7.21).

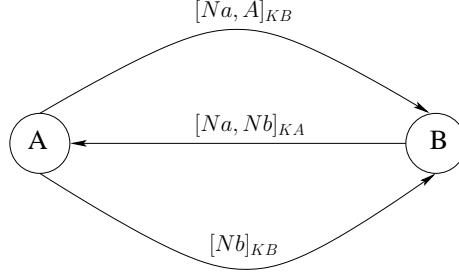


Figure 7.21: Needham-Schröder protocol

1. A person A decides to communicate with a person B and sends to B a nonce Na encoded with a public key of B (KB).
2. Person B generates another nonce Nb and sends it together with Na back to A encoded with a public key of A (KA). Now A is sure of the identity of B .
3. In the last step A sends to B the nonce Nb encoded with the public key of B (KB). Now B is also sure of the identity of A .

In our model of the Needham-Schröder protocol we consider a fixed number of persons having unique ids. One chosen person is allowed to act additionally as an intruder. The intruder can resend an obtained message to another as the addressee or re-encode it if the message is encoded with the intruder’s public key. (These are so called replay attacks). During the execution of a protocol we generate nonces which play an important role in the authentication procedure. Nonces are in general random numbers from some interval, but here we consider the simplified treatment of nodes in order only to make them unique in a sense described below. We use a pair (id_1, id_2) as nonce for a participant with id_1 sending a message to a participant with id_2 . We say that an intruder with id_i has successfully attacked the protocol if the intruder can decode a message containing a nonce (id_1, id_2) , where $id_i \neq id_1$ and $id_i \neq id_2$.

This approach gives us unique ids for each pair of communicating partners, but it does not allow us to distinguish between two different sessions for the same participant. The latter could be reached for example if we assign a unique id (id_s) to each established session and use tuples (id_1, id_2, id_s) as corresponding nonces. Also, the definition of an attack is here rather special and can be extended in different ways: For example an intruder could modify the open (non-encoded) parts of the message arbitrarily. The definition here is exactly the one we need in order to model the attack from [75] with an AGTS.

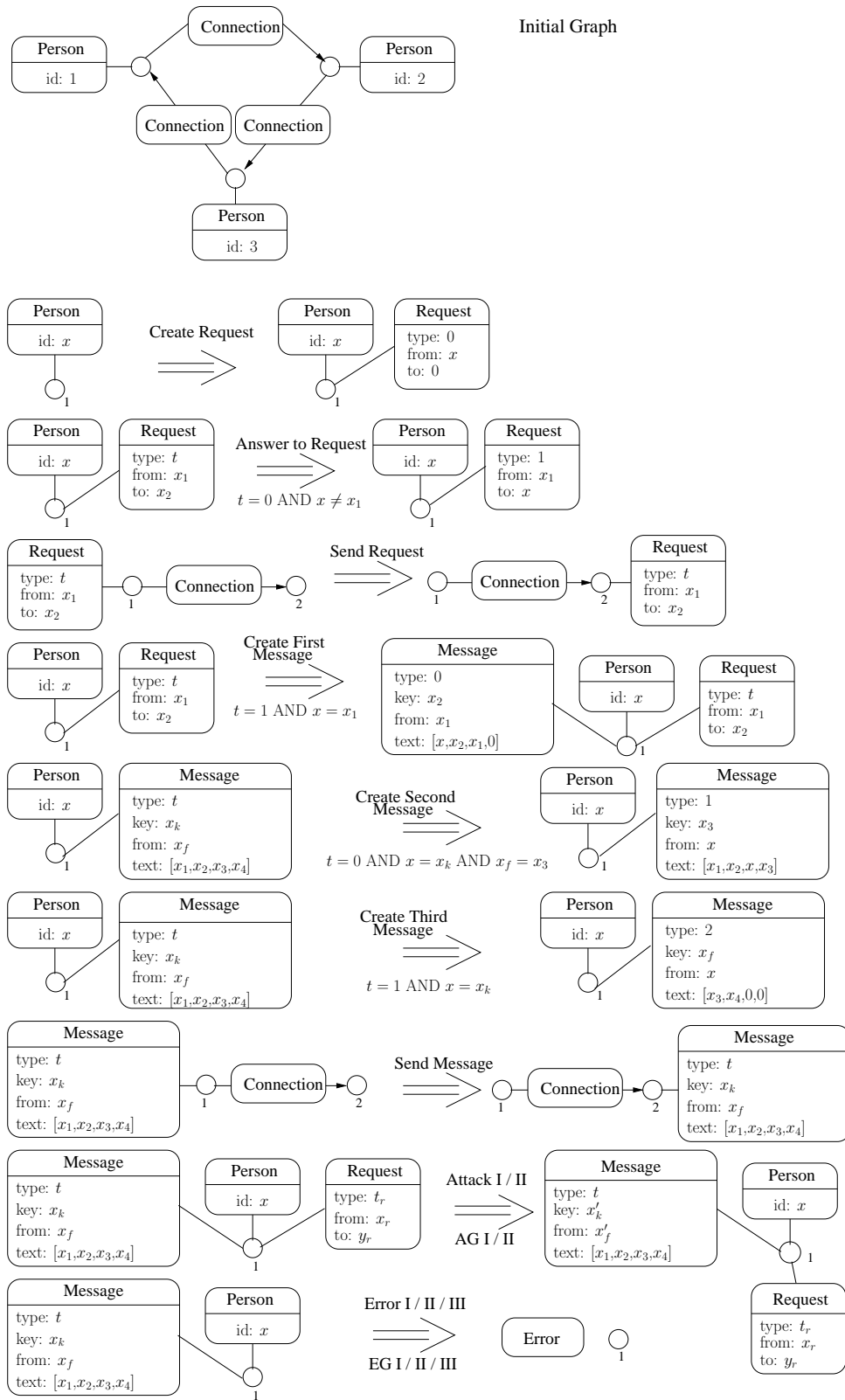


Figure 7.22: AGTS model of Needham-Schröder protocol

In Fig. 7.22 there is an AGTS specification of the Needham-Schröder protocol with an intruder. Although the number of communicating persons is limited to three, the system is still infinite because of the non-limited number of created messages.

In the first stage of the protocol a handshake between participants is made. This is achieved by rules “Create Request”, “Send Request” and “Answer Request”. The edge “Request” has integer attributes “from” and “to” and has a twofold purpose: with the help of the “Request” edge a person can establish contact to another person and start the authentication procedure. Also the intruder can use this edge in order to get information about other participants.

The next stage of the protocol is the sending of three encoded messages. A message is represented by the edge “Message” and has the following attributes: *key*: id of a person whose public key is used to encode the message; *type*: one of the three types of messages; *from*: id of the sender; *text*: tuple representing the content of the message. The content of a message is represented in the following way: for example for two nonces $Na = (id_1^A, id_2^A)$ and $Nb = (id_1^B, id_2^B)$ the message content $[Na, Nb]$ is a tuple $[id_1^A, id_2^A, id_1^B, id_2^B]$.

The rules “Create First Message”, “Create Second Message” and “Create Third Message” describe the creation of messages and rule “Send Message” describes the sending of a message via a connection.

For an attack the intruder has the following possibilities: If the intruder can decode the message, then the first possibility is to encode it with another person’s public key. The second possibility is to change the field “from” in the message.

We use the following guard expressions in the rules:

$$AGI = ((x_k = x) \wedge (t_r = 1) \wedge (x = 3))$$

$$AGII = ((x_f \neq x) \wedge (t_r = 1) \wedge (x = 3)).$$

AGI means that the third person (intruder) can decode the message and also has obtained an acknowledgement to his request. In this case $x'_k = y_r$ and $x'_f = x_f$. Now the intruder can send the message to y_r , who will consider it as a message from x_f .

In the case of *AGII* the intruder $x = 3$ only re-sends the message from x_f , not equal to the intruder, without decoding it. Here $x'_k = x_k$ and $x'_f = x$, i.e., the intruder supposes that the message is from him.

As declared earlier, an error has occurred if the intruder captures another participant’s nonce. We have three different error rules corresponding to attacks on three different types of messages. Corresponding guard expressions describe that the person is an intruder and some nonces in the captured messages belong to another person.

$$EGI = ((x \neq x_1) \wedge (x \neq x_2) \wedge (x_1 \neq 0) \wedge (x = x_k) \wedge (t = 0) \wedge (x = 3))$$

$$EGII = (((x \neq x_1) \wedge (x \neq x_2) \wedge (x_1 \neq 0) \wedge (x_2 \neq 0)) \vee ((x \neq x_3) \wedge (x \neq x_4) \wedge (x_3 \neq 0) \wedge (x_4 \neq 0))) \wedge (x = x_k) \wedge (t = 1) \wedge (x = 3))$$

$$EGIII = ((x \neq x_1) \wedge (x \neq x_2) \wedge (x_1 \neq 0) \wedge (x_2 \neq 0) \wedge (x = k) \wedge (t = 2) \wedge (x = 3))$$

For example, *EGI* means that the intruder ($x = 3$) can decode ($x = x_k$) the first message ($t = 0$) and the message is from x_1 to x_2 with $(x \neq x_1) \wedge (x \neq x_2)$. In this case the intruder captures a nonce $x_1 \neq 0$ which belongs to another person.

The property to be verified is: “No Error edge will be created”, i.e., the protocol is free from attacks.

We should decide which abstraction of attributes should be used. Modulo abstraction as in the second example is not appropriate here because it cannot give us an answer to the question if $x = y$ or $x \neq y$, which is needed for the evaluation of the guard expressions. In our experiments we have chosen interval abstraction.

Now we can unfold the AGTS of the Needham-Schröder protocol and analyse it using coverability techniques. In our experiments we have mostly used the coverability graph because the backward reachability procedure with attributes has a strong memory explosion with the growth of parameters in the interval abstraction.

In the first over-approximation all connections and persons are merged and the obtained counter-example is structurally too coarse. After the refinement step we obtain another counter-example which is structurally real but is still spurious because of the abstraction of attributes. This is the case until we use the interval $[0, 3]$ in the interval abstraction (we start with the interval $[0, 1]$ and increase the right bound by one at each step). After this we obtain the real counter-example which corresponds to the attack found by Löwe in [75]. This is a kind of a man-in-the-middle attack, where an intruder persuades one person (A) to initiate a session with him and then resends the messages to another person (B). In this way the intruder convinces B that B communicates with A.

Chapter 8

Conclusion and Future Work

In this thesis we discussed the verification of (attributed) graph transformation systems by over-approximating them with (attributed) Petri graphs which are (attributed) Petri nets having additional hypergraph structure. We consider both graphs that are abstracted by merging nodes and edges (using the concept of graph morphisms) and the abstraction of data values based on the abstract interpretation approach. The results of the work can be summarized as follows:

- The counterexample-guided abstraction refinement technique was developed for graph transformation systems.
- A verification technique based on over-approximations (included also the counterexample-guided abstraction refinement) was developed for attributed graph transformation systems.
- The verification tool AUGUR 2 supporting the whole verification process for (attributed) graph transformation systems was designed and implemented.
- With AUGUR 2 a number of case studies were successfully solved. Besides concrete attributed and non-attributed systems we also gathered some statistical material.
- For the optimization of the approximation procedure the incremental coverability approach for Petri nets was developed. The technique is rather general and we believe it can also be useful for other applications where Petri nets are updated interactively.

We believe that introducing counterexample-based abstraction refinement for both attributed and non-attributed graph transformation systems is an important step in order to make such verification techniques usable in practice. We also think that some of the techniques presented here can be employed in fairly general settings.

We also hope that AUGUR 2 will be used in the future for the verification of graph transformation systems and new algorithms optimizing the verification procedure will be added.

Unfortunately the attribute abstraction based on the abstract interpretation gives no guarantee that the spurious counter-example will be eliminated. Future work in this case is the development and the implementation of the *predicate abstraction* technique [49, 41, 50, 59] for Petri graphs and AGTSs. This technique uses predicates as an abstraction of data types and allows one to choose the necessary number of predicates in order to eliminate the given counterexample. The technique of predicate abstraction was

developed for standard program languages and transferring it to the Petri net semantic is a non-trivial task.

In our setting the difficulty is not so much how to generate these predicates (after all, we have a specific counterexample) but how to interpret them over the markings of the Petri net. The situation would be easy if all predicates were unary, since in this case we would employ the concept of Galois connections. However generated predicates typically have a higher arity, typically predicates are binary predicates of the form $\mathbf{x} < \mathbf{y}$. For the original predicate abstraction approach this is not a problem since there are only finitely many variables and the value of predicates for an abstract state can be described in a finite way. However in our case there can be arbitrarily many tokens and it is not clear to us how to solve the coverability problem for Petri nets with such an abstraction mechanism.

Another direction for future work could be the verification of more practically oriented case studies with more details. As an example we mention here design patterns for J2EE [30], where the specified systems have both dynamic behaviour, which can be modelled by graph transformations, and data-flows, which can be compactly described using attributes. Many of the introduced design patterns are rather new and formal verification of them could be very useful.

Bibliography

- [1] Parosh Aziz Abdulla, Bengt Jonsson, Mats Kindahl, and Doron Peled. A general approach to partial order reductions in symbolic verification. In *Proc. of CAV '98*, pages 379–390. Springer, 1998. LNCS 1427.
- [2] S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum. *Handbook of Logic in Computer Science: Volume 4*. Clarendon Press, 1995.
- [3] Jiří Adámek, Horst Herrlich, and George Strecker. *Abstract and concrete categories*. Wiley-Interscience, New York, NY, USA, 1990.
- [4] P. Baldan and B. König. Approximating the behaviour of graph transformation systems. In *Proc. of ICGT '02*, pages 14–29. Springer, 2002. LNCS 2505.
- [5] Paolo Baldan, Andrea Corradini, Javier Esparza, Tobias Heindel, Barbara König, and Vitali Kozioura. Verifying red-black trees. In *Proc. of COSMICA'05*, 2005. Proceedings available as report RR-05-04 (Queen Mary, University of London).
- [6] Paolo Baldan, Andrea Corradini, and Barbara König. A static analysis technique for graph transformation systems. In *Proc. of CONCUR '01*, pages 381–395. Springer-Verlag, 2001. LNCS 2154.
- [7] Paolo Baldan, Andrea Corradini, and Barbara König. Static analysis of distributed systems with mobility specified by graph grammars—a case study. In H. Ehrig, B. Krämer, and A. Ertas, editors, *Proc. of IDPT '02 (Sixth International Conference on Integrated Design & Process Technology)*. Society for Design and Process Science, 2002.
- [8] Paolo Baldan, Andrea Corradini, and Barbara König. Unfolding-based verification for graph transformation systems. In *Proc. of UniGra '03: Uniform Approaches to Graphical Specification Techniques (Warsaw)*, 2003.
- [9] Paolo Baldan, Andrea Corradini, and Barbara König. Verifying finite-state graph grammars: an unfolding-based approach. In *Proc. of CONCUR '04*, pages 83–98. Springer-Verlag, 2004. LNCS 3170.
- [10] Paolo Baldan, Andrea Corradini, Barbara König, and Alberto Lluch Lafuente. A temporal graph logic for verification of graph transformation systems. In *Proc. of WADT '06 (Workshop on Algebraic Development Techniques)*, pages 1–20. Springer, 2007. LNCS 4409.
- [11] Paolo Baldan, Andrea Corradini, and Ugo Montanari. Unfolding and event structure semantics for graph grammars. In *Proc. of FoSSaCS '99*, pages 73–89, 1999.

- [12] Paolo Baldan and Barbara König. Approximating the behaviour of graph transformation systems. In *Proc. of ICGT '02 (International Conference on Graph Transformation)*, pages 14–29. Springer-Verlag, 2002. LNCS 2505.
- [13] Paolo Baldan, Barbara König, and Bernhard König. A logic for analyzing abstractions of graph transformation systems. In *Proc. of SAS '03 (International Static Analysis Symposium)*, pages 255–272. Springer-Verlag, 2003. LNCS 2694.
- [14] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *International Workshop on SPIN Model Checking '01*, pages 103–122. Springer, 2001. LNCS 2057.
- [15] Roswitha Bardohl. GenGED - a generic graphical editor for visual languages based on algebraic graph grammars. In *in Proceedings of the IEEE Symposium on Visual Languages VL '98*, page 48. IEEE Computer Society, 1998.
- [16] Julian Bart. Effiziente Entfaltungsalgorithmen für Graphersetzungssysteme. Master's thesis, Universität Stuttgart, June 2005. No. 2290.
- [17] Gernot Veit Batz. An optimization technique for subgraph matching strategies. Technical Report 2006-7, Universität Karlsruhe, IPD Goos, April 2006.
- [18] Jörg Bauer. *Analysis of Communication Topologies by Partner Abstraction. PhD thesis*. Verlag Pirrot, Saarbrücken, 2006.
- [19] Jörg Bauer, Iovka Boneva, Marcos E. Kurbán, and Arend Rensink. A modal-logic based graph abstraction. In *Proc. of ICGT '08*, pages 321–335. Springer, 2008. LNCS 5214.
- [20] Marek A. Bednarczyk and Andrzej M. Borzyszkowski. General morphisms of petri nets (extended abstract). In *Proc. of ICALP*, pages 190–199. Springer, 1999. LNCS 1644.
- [21] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *Proc. of LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, 2000.
- [22] Francis Borceux. *Handbook of Categorical Algebra 3 (Categories of Sheaves)*, volume 52. Cambridge University Press, January 1995.
- [23] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *Proc. of ICSE '03 (25th International Conference on Software Engineering)*, pages 385–395. IEEE Computer Society, 2003.
- [24] E. Clarke, S. Grumberg, S. Jha, and H. Lu, Y. und Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV 2000*, pages 154–169. Springer, 2000. LNCS 1855.
- [25] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. 16(5):1512–1542, 1994.
- [26] Edmund M. Clarke and Bernd-Holger Schlingloff. Model checking. In *Handbook of Automated Reasoning*, pages 1635–1790. 2001.

- [27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2001. Second Edition.
- [28] P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.
- [29] Patrick Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, 1996.
- [30] William Crawford and Jonathan Kaplan. *J2EE Design Patterns*. O'REILLY©, 2003.
- [31] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *Proc. of CAV'99*, pages 160–171, 1999.
- [32] Fernando Luís Dotti, Luciana Foss, Leila Ribeiro, and Osmar Marchi Santos. Verification of distributed object-based systems. In *Proc. of FMOODS '03*, pages 261–275. Springer, 2003. LNCS 2884.
- [33] Fernando Luís Dotti, Barbara König, Osmar Marchi dos Santos, and Leila Ribeiro. A case study: Verifying a mutual exclusion protocol with process creation using graph transformation systems. Technical Report 08/2004, Universität Stuttgart, 2004.
- [34] Stefan Edelkamp and Jörg Hoffmann. PDDL2.2: The language for the classical part of the 4th international planning competition, january 21. Technical Report 195, 2004.
- [35] Stefan Edelkamp and Shahid Jabbar. Action planning for directed model checking of Petri nets. In *Proc. of MoChArt 2005*, volume 149(2), pages 3–18, 2006. ENTCS.
- [36] Stefan Edelkamp and Mario Lischka. Heuristic search planning in administration nets. Technical report, Computer Science Department, University of Dortmund, Germany. May 25, 2005.
- [37] Hartmut Ehrig. Behaviour and instantiation of high-level petri net processes. *Fundam. Inform.*, 65(3):211–247, 2005.
- [38] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graphs and graph transformation based on adhesive HLR categories. In *Fundamenta Informaticae, Volume 74, Issue 1*, pages 31–61. IOS Press, 2006.
- [39] Hartmut Ehrig, Julia Padberg, and Leila Ribeiro. Algebraic high level nets: Petri nets revisited. In *Selected papers from 9th workshop on Specification of abstract data types: recent trends in data type specification*, pages 188–206, Secaucus, NJ, USA, 1992. Springer-Verlag New York, Inc.
- [40] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graph transformation. In *Proc. of ICGT '04*, pages 161–177. Springer, 2004. LNCS 3256.
- [41] Javier Esparza, Stefan Kiefer, and Stefan Schwoon. Abstraction refinement with craig interpolation and symbolic pushdown systems. In *Proc. of TACAS '06*, pages 489–503. Springer, 2006. LNCS 3920.

- [42] Javier Esparza, Stefan Rüdiger, and Walter Vogler. An improvement of McMillan's unfolding algorithm. In *Proc. of TACAS'96*, pages 87–106, 1996.
- [43] A. Finkel, G. Geeraerts, J.-F. Raskin, and L. Van Begin. A counter-example to the minimal coverability tree algorithm. Technical Report 535, Université Libre de Bruxelles, 2005.
- [44] Alain Finkel. The minimal coverability graph for Petri Nets. In *Proc. of ATPN (Applications and Theory of Petri Nets)*, pages 210–243. Springer, 1991. LNCS 674.
- [45] Alain Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.
- [46] Fabio Gadducci, Reiko Heckel, and Manuel Koch. A fully abstract model for graph-interpreted temporal logic. In *Proc. of TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, pages 310–322, London, UK, 2000. Springer-Verlag.
- [47] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Proc. of CAV'97*, pages 72–83, 1997.
- [48] Reiko Heckel. Compositional verification of reactive systems specified by graph transformation. In *In Proc. of FASE*, pages 138–153, 1998.
- [49] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proc. of POPL'04*, pages 232–244. ACM, 2004.
- [50] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proc. of POPL '02*, pages 58–70. ACM, 2002.
- [51] Jörg Hoffmann. The Metric-FF planing system: Translation ignoring delete lists to numeric state variables. *Journal of Artificial Intelligence Research*, 20:291–341, 2003.
- [52] Kathrin Hoffmann, Hartmut Ehrig, and Till Mossakowski. High-level nets with nets and rules as tokens. In *Proc. of ICATPN 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 268–288. Springer, 2005.
- [53] Richard C. Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. Gxl: A graph-based standard exchange format for reengineering. *Science of Computer Programming*, 60(2):149–170, 4 2006.
- [54] Markus Holzer and Barbara König. Regular languages, sizes of syntactic monoids, graph colouring, state complexity results, and how these topics are related to each other. *EATCS Bulletin*, 83:139–155, June 2004. Appeared in The Formal Language Theory Column.
- [55] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.
- [56] Martin Horsch. Test case generation for rule-based translators. Master's thesis, Universität Stuttgart, June 2005. Studienarbeit (Student research project), No. 1984.
- [57] Kurt Jensen. Coloured Petri Nets. In *Advances in Petri Nets*, pages 248–299, 1986.
- [58] Kurt Jensen. Coloured Petri Nets: Status and outlook. In *In Proc. of ICATPN*, pages 1–2, 2003.

- [59] Ranjit Jhala and K.L. McMillan. A practical and complete approach to predicate refinement. In *Proc. of TACAS'06*, pages 459–473. Springer, 2006. LNCS 3920.
- [60] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969.
- [61] Harmen Kastenbergh. Towards attributed graphs in GROOVE. In *Proceedings of Workshop on Graph Transformation for Verification and Concurrency*, volume 05-34 of *CTIT Technical Report*, pages 91–98, 2005.
- [62] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, 1992.
- [63] Manuel Koch. *Integration of Graph Transformation and Temporal Logic for the Specification of Distributed Systems. PhD thesis*. Technische Universität Berlin, 2000.
- [64] Barbara König. A general framework for types in graph rewriting. *Acta Informatica*, 42(4-5):349–388, 2005.
- [65] Barbara König. Graph transformation systems, Petri nets and semilinear sets: Checking for the absence of forbidden paths in graphs. In *Proc. of PNGT '06 (Workshop on Petri Nets and Graph Transformation)*, volume 2 of *Electronic Communications of the EASST*, 2007.
- [66] Barbara König and Vitali Kozioura. Augur—a tool for the analysis of graph transformation systems. *EATCS Bulletin*, 87:125–137, November 2005. Appeared in The Formal Specification Column.
- [67] Barbara König and Vitali Kozioura. Augur 2—a new version of a tool for the analysis of graph transformation systems. In *Proc. of GT-VMT '06 (Workshop on Graph Transformation and Visual Modeling Techniques)*, pages 201–210, 2006. ENTCS, Volume 211.
- [68] Barbara König and Vitali Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In *Proc. of TACAS '06*, pages 197–211. Springer, 2006. LNCS 3920.
- [69] Barbara König and Vitali Kozioura. Incremental construction of coverability graphs. *Information Processing Letters*, 103(5):203–209, 2007.
- [70] Barbara König and Vitali Kozioura. Towards the verification of attributed graph transformation systems. In *Proc. of ICGT '08 (International Conference on Graph Transformation)*. Springer, 2008. LNCS, to appear.
- [71] Vitali Kozioura. Verification of random graph transformation systems. In *Proc. of GT-VC 2006 (Workshop on Graph Transformation for Concurrency and Verification)*, volume 175(4), pages 63–72, 2007. ENTCS.
- [72] Leen Lambers. A new version of GTXL: An exchange format for graph transformation systems. In *Proc. Workshop on Graph-Based Tools (GraBaTs'04)*, Electronic Notes in Theoretical Computer Science, pages 51–63, 2004.
- [73] Saunders M. Lane. *Categories for the Working Mathematician (Graduate Texts in Mathematics)*. Springer, September 1998.
- [74] Javier Larrosa and Gabriel Valiente. Constraint satisfaction algorithms for graph pattern matching. *Mathematical Structures in Computer Science*, 12(4):403–422, 2002.

- [75] Gavin Loewe. An attack on the Needham-Schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56(3):131–133, 1995.
- [76] A. Loginov, T. Reps, and M. Sagiv. Abstraction refinement for 3-valued-logic analysis. Technical Report 1504, Comp. Sci. Dept., Univ. of Wisconsin, 2004.
- [77] M. Löwe and H. Ehrig. Algebraic approach to graph transformation based on single pushout derivations. In *16th Int. Workshop on Graph Theoretic Concepts in Computer Science*, pages 338–353. Springer, 1991. LNCS 484.
- [78] Michael Löwe, Martin Korff, and Annika Wagner. An algebraic framework for the transformation of attributed graphs. In *Term graph rewriting: theory and practice*, pages 185–199. John Wiley and Sons Ltd., 1993.
- [79] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc, 1996.
- [80] Ernst W. Mayr. An algorithm for the general Petri net reachability problem. *SIAM J. Comput.*, 13(3):441–460, 1984.
- [81] K.L. McMillan. *Symbolic Model Checking*. PhD thesis. University of Kluwer, 1993.
- [82] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. EATCS Monographs on Theoretical Computer Science. 1984.
- [83] M. Minas. Specifying diagram languages by means of hypergraph grammars. In *Proc. Thinking with Diagrams (TwD’98)*, pages 151–157, 1998.
- [84] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.
- [85] W. Reisig. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1985.
- [86] Nicolas Relange. Verifikation dynamischer Systeme: Reguläre Ausdrücke zur Spezifikation verbotener Pfade. Master’s thesis, Universität Stuttgart, September 2004. No. 2192.
- [87] Arend Rensink. Towards Model Checking Graph Grammars. In *Proc. of AVOCS ’03 (Workshop on Automated Verification of Critical Systems)*, pages 150–160. University of Southampton, 2003.
- [88] Arend Rensink. Canonical graph shapes. In *Proc. of ESOP ’04*, pages 401–415. Springer, 2004. LNCS 2986.
- [89] Arend Rensink. State space abstraction using shape graphs. In *Proc. of AVIS ’04 (Third International Workshop on Automatic Verification of Infinite-State Systems)*, ENTCS, 2004.
- [90] Arend Rensink. Isomorphism checking in GROOVE. *Electronic Communications of the EASST: Graph Based Tools*, 1, 2006.
- [91] Arend Rensink and Dino Distefano. Abstract graph transformation. In *International Workshop on Software Verification and Validation (SVV)*, Electronic Notes in Theoretical Computer Science, 2005.
- [92] Arend Rensink and Dániel Varró. Model checking graph transformations: A comparison of two approaches. In *Proc. of ICGT ’04*, pages 226–241. Springer, 2004. LNCS 3256.

- [93] Christophe Reutenauer. *Aspects mathématiques des réseaux de Pétri*. Masson, 1989.
- [94] L. Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars*. *PhD thesis*. Technische Universität Berlin, 1996.
- [95] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, volume 1. World Scientific, 1997.
- [96] Michael Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In *In Proc. of TAGT '98*, pages 238–251. Springer-Verlag, 1998. LNCS 1764.
- [97] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Symposium on Principles of Programming Languages*, pages 105–118, 1999.
- [98] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS (ACM Transactions on Programming Languages and Systems)*, 24(3):217–298, 2002.
- [99] Karsten Schmidt. Model-checking with coverability graphs. *Formal Methods in System Design*, 15(3):239–254, 1999.
- [100] Karsten Schmidt. LoLA: A low level analyser. In *Proc. of ATPN (Application and Theory of Petri Nets)*, pages 465–474. Springer, 2000. LNCS 1825.
- [101] A. Schürr. PROGRES: A VHL-language based on graph grammars. In *in Proc. 4th Int. Workshop on Graph-Grammars and Their Application to Computer Science*, pages 641–659. Springer, 1991. LNCS 532.
- [102] Zündorf A. Schürr A., Winter A. Graph grammar engineering with PROGRESS. In *Proc. of the 5th European Software Engineering Conference (ESEC'95)*, pages 219–234. Springer, 1995. LNCS 989.
- [103] Joseph Sifakis. Property preserving homomorphisms of transition systems. In *Proceedings of the Carnegie Mellon Workshop on Logic of Programs*, pages 458–473. Springer-Verlag, 1983.
- [104] Gabriele Taentzer. AGG: A tool environment for algebraic graph transformation. In *Proc. of AGTIVE '99 (Applications of Graph Transformations with Industrial Relevance, International Workshop)*, pages 481–488. Springer, 1999. LNCS 1779.
- [105] Timur Tsothniashvili. Übersetzung von imperativen Programmen mit Zeigermanipulation in Graphtransformations-Regeln. Master's thesis, Universität Stuttgart, June 2006. No. 2431.
- [106] Sinan Turan. Effiziente Berechnung der Überdeckbarkeit bei Petri-Netzen. Master's thesis, Universität Stuttgart, June 2004. Studienarbeit (Student research project), No. 1935.
- [107] Rüdiger Valk and Guy Vidal-Naquet. Petri Nets and regular languages. *Journal of Computer and System Sciences*, 23(3):299–325, 1981.
- [108] Dániel Varró. Towards symbolic analysis of visual modeling languages. In *Workshop on Graph Transformation and Visual Modeling Techniques '02*, volume 72 of *ENTCS*, pages 57–70. Elsevier, 2002.

- [109] Gergely Varró and Dániel Varró. Graph transformation with incremental updates. In *Proc. GT-VMT 2004, International Workshop on Graph Transformation and Visual Modelling Techniques*, volume 109 of *ENTCS*, pages 71–83. Elsevier, 2004.
- [110] Gergely Varró, Dániel Varró, and Katalin Friedl. Adaptive graph pattern matching for model transformations using model-sensitive search plans. In G. Karsai and G. Taentzer, editors, *Proc. of GraMot 2005, International Workshop on Graph and Model Transformations*, ENTCS, pages 191–205. Elsevier, 2006.
- [111] Arwed von Merkat. Analyse von Graphtransformationssystemen mit Hilfe von Petrinetzen und Logiken. Master’s thesis, Universität Stuttgart, July 2006. No. 2442.
- [112] Ingo Walther. Implementation of an algorithm for the analysis of graph transformation systems. Master’s thesis, Technische Universität München, December 2003. Systementwicklungsprojekt (Student research project).
- [113] Glynn Winskel. A new definition of morphism on Petri nets. In *Proc. of STACS ’84*, pages 140–150. Springer, 1984. LNCS 166.
- [114] Andreas Winter, Bernt Kullbach, and Volker Riediger. An overview of the GXL graph exchange language. In *S. Diehl (ed.) Software Visualization International Seminar Dagstuhl Castle, Germany, May 20-25, 2001 Revised Lectures*, 2002.
- [115] Martin Wirsing. Algebraic specification. In Jan van Leeuwen, editor, *Formal Models and Semantics, Handbook of Theoretical Computer Science*, volume B, chapter 13. Elsevier, 1990.
- [116] Maxim Zaks. Effiziente Algorithmen für die Analyse von Graphtransformationssystemen. Master’s thesis, Universität Dortmund, March 2007.
- [117] Albert Zündorf. Graph pattern matching in PROGRES. In *In Proc. of TAGT ’94*, pages 454–468. Springer-Verlag, 1994. LNCS 1073.

Appendix A

Basic Category Theory for Graph Rewriting

In this section we give a brief introduction to category theory and its applications to graph rewriting. A detailed description of category theory can be found in [3, 73, 22] and applications to graph rewriting in [95, 12].

Category theory represents in an abstract way mathematical objects and relationships between them. Categories now appear in most branches of mathematics and in some areas of theoretical computer science. Here we show how graph rewriting and the approximation of GTSs by Petri graphs (unfolding and folding steps) can be represented in the framework of category theory.

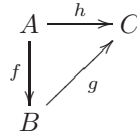
We start with the definition of a category.

Definition A.0.1 (category) *A category Cat consists of a class $ob(Cat)$ of objects, a class $hom(Cat)$ of morphisms and a binary operation \circ , called composition of morphisms. Each morphism f has a unique source object A and target object B and we write in this case $f : A \rightarrow B$ and say "f is a morphism from A to B ". We denote by $Hom(A, B)$ the class of all morphisms from A to B . For the binary operation \circ and any three objects A , B , and C , we have $\circ : Hom(A, B) \times Hom(B, C) \rightarrow Hom(A, C)$. The composition of $f : A \rightarrow B$ and $g : B \rightarrow C$ is written as $g \circ f$.*

The following axioms hold:

1. *Associativity: If $f : A \rightarrow B$, $g : B \rightarrow C$ and $h : C \rightarrow D$ then $h \circ (g \circ f) = (h \circ g) \circ f$.*
2. *Identity: For every object X , there exists a morphism $id_X : X \rightarrow X$ called the identity morphism for X , such that for every morphism $f : A \rightarrow B$, we have $id_B \circ f = f = f \circ id_A$.*

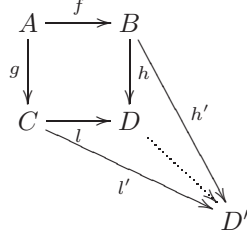
From the axioms, it can be proved that there is exactly one identity morphism for every object. Relations among morphisms are often depicted using commutative diagrams. For example $g \circ f = h$ can be depicted as



Here vertices represent objects and arrows represent morphisms. The influence of commutative diagrams has been such that "arrow" and morphism are now synonymous. Morphisms are typically structure-preserving maps.

In order to describe graph rewriting in the framework of category theory we need the definition of a pushout (application of a rewriting rule to a hypergraph is formalized in the DPO – double pushout–approach).

Definition A.0.2 (pushout) *Given a category Cat and two arrows $f : A \rightarrow B$ and $g : A \rightarrow C$ of C , a triple $(D, h : B \rightarrow D, l : C \rightarrow D)$ as in the diagram below is called a pushout of (f, g) if*



1. $h \circ f = l \circ g$, and
2. for all objects d' and arrows $h' : B \rightarrow D'$ and $l' : C \rightarrow D'$ such that $h' \circ f = l' \circ g$, there exists a unique arrow $k : D \rightarrow D'$ such that $k \circ h = h'$ and $k \circ l = l'$.

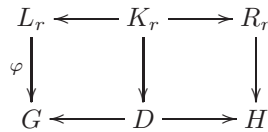
The category **Graph** is a category having hypergraphs as objects, morphisms between hypergraph as arrows and the composition of morphisms is denoted by \circ . For the description of graph rewriting we use rules as in the double-pushout approach [3], with the same restrictions as in Chapter 3.

Definition A.0.3 (rewriting rule) *A graph rewriting rule is a span of injective hypergraph morphisms $r = (L_r \xleftarrow{\varphi_L} K_r \xrightarrow{\varphi_R} R_r)$, where L_r, K_r, R_r are finite hypergraphs.*

In Chapter 3 we have introduced a rewriting rule having some restrictions (Definition 3.1.2). We formulate here these restrictions:

1. K_r is discrete, i.e. it contains no edges
2. The morphism φ_L is bijective on nodes
3. V_{L_r} does not contain isolated nodes.

Definition A.0.4 (hypergraph rewriting) *Let r be a rewriting rule. A match of r in a hypergraph G is any morphism $\varphi : L_r \rightarrow G$. In this case we write $G \Rightarrow_r H$, if there exists a double-pushout diagram*



Because of the restrictions above the morphisms $D \rightarrow G$ is injective. $D \rightarrow H$ is also injective if $K_r \rightarrow R_r$ is injective.

This definition coincides with the definition of a rewriting step from Chapter 3 (see Definition 3.1.3). Before we define the category of Petri graphs we recall here the definition of a Petri graph (see Chapter 3, Definition 3.2.1). Petri graphs are used to represent an over-approximation of GTSS. Note that the edges of the graph are at the same time the places of the net and that the transitions are labelled with rules of the GTS.

Definition A.0.5 (Petri graph) Let $\mathcal{G} = (\mathcal{R}, G_0)$ be a GTS. A Petri graph (over \mathcal{R}) is a tuple $P = (G, N, \mu)$, where G is a hypergraph, $N = (E_G, T_N, \bullet(), ()^\bullet, p_N)$ is an \mathcal{R} -labelled Petri net (the initial marking m_0 is defined below) where the places are the edges of G and μ associates to each transition $t \in T_N$, with $p_N(t) = (L, R, id)$, a hypergraph morphism $\mu(t) : L \cup R \rightarrow G$ such that $\bullet t = \mu(t)^\oplus(E_L)$ and $t^\bullet = \mu(t)^\oplus(E_R)$.

A Petri graph for the GTS \mathcal{G} is a pair (P, ι) , where $P = (G, N, \mu)$ is a Petri graph over \mathcal{R} and $\iota : G_0 \rightarrow G$ is a graph morphism. A marking is reachable (coverable) in Petri graph if it is reachable (coverable) in the underlying Petri net with the multiset $\iota^\oplus(E_{G_0})$ as the initial marking m_0 .

We view Petri graphs as symbolic representations of transition systems with graphs as states. Now we define the category of Petri graphs. As it was described in Chapter 3 Petri graphs are used as over-approximations of GTSs.

Definition A.0.6 (category of Petri graphs) A Petri graph morphism is a pair $\psi = (\varphi, \tau) : (G, N, \mu) \rightarrow (G', N', \mu')$, where

1. $\varphi : G \rightarrow G'$ is a hypergraph morphism.
2. $\tau : T_N \rightarrow T_{N'}$ is a mapping such that for every $t \in T_N$, $\bullet \tau(t) = \varphi^\oplus(\bullet t)$ and $\tau(t)^\bullet = \varphi^\oplus(t^\bullet)$, and $p_{N'} \circ \tau = p_N$.
3. For every $t \in T_N$, $\mu'(\tau(t)) = \varphi \circ \mu(t)$.

The category of Petri graphs and Petri graph morphisms is denoted by **PG**.

Now we can define unfolding and folding steps used for the approximation of GTSs by Petri graphs in the framework of category theory. In this way we can speak about the whole approximation procedure in categorical terms.

To formally describe an unfolding step we need to fix some notation. Given a transition t and a rule r we will denote by $P(t, r)$ the Petri graph $(L_r \cup R_r, N, \mu)$ where $N = (E_{L_r \cup R_r}, \{t\}, \bullet t = E_{L_r}, t^\bullet = E_{R_r}, p_N(t) = r)$ and $\mu(t) = id_{L_r \cup R_r}$. By \emptyset we denote a function with an empty set as domain. The following construction is equivalent to the unfolding step in Construction 3.2.7.

Proposition A.0.7 (unfolding operation) Let $\mathcal{P} = (G, N, \mu)$ be a Petri graph for a GTS, let r be a rule and let $\varphi : L_r \rightarrow G$ be a match of r in G . The unfolding of \mathcal{P} with rule r at match φ (denoted by $unf(\mathcal{P}, r, \varphi)$) is the Petri graph obtained as the pushout of $(\varphi, \emptyset) : [L_r] \rightarrow \mathcal{P}$ and $(id_{L_r}, \emptyset) : [L_r] \rightarrow P(t, r)$.

Before we describe the folding operation we need the notion of a *coequalizer* from category theory. A coequalizer is a generalization of a quotient by an equivalence relation on objects.

Definition A.0.8 (coequalizer) Let X and Y be two objects and $f, g : X \rightarrow Y$ are two parallel morphisms. A coequalizer is defined as an object Q together with a morphism $q : Y \rightarrow Q$ such that $q \circ f = q \circ g$. Moreover, the pair (Q, q) must be universal in the sense that given any other such pair (Q', q') , there exists a unique morphism $u : Q \rightarrow Q'$ for which the following diagram commutes:

$$\begin{array}{ccccc} X & \xrightleftharpoons[g]{f} & Y & \xrightarrow{q} & Q \\ & & \searrow q' & \downarrow u & \\ & & & Q' & \end{array}$$

Now we are ready to define the folding step. The following construction is equivalent to the folding step in Construction 3.2.7.

Proposition A.0.9 (folding operation) *Let $\mathcal{P} = (G, N, \mu)$ be a Petri graph for a GTS. Let r be a rule of the GTS and let $\varphi', \varphi : L_r \rightarrow G$ be matches of r in G . The folding of \mathcal{P} at the matches φ', φ (denoted $\text{fold}(\mathcal{P}, r, \varphi', \varphi) = \mathcal{P}'$) is the Petri graph \mathcal{P}' obtained as the coequalizer of $(\varphi, \emptyset), (\varphi', \emptyset) : [L_r] \rightarrow \mathcal{P}$ in category \mathbf{PG} .*

Appendix B

Example in GTXL format

In this section we present a commented description of an AGTS from Section 4.2 (Fig. 4.3) in GTXL format. The example consists of an initial graph, which is one edge labelled "A" with two attributes "a1" and "a2", and one rewriting rule.

```
<?xml version="1.0"?>
<!DOCTYPE gtxl SYSTEM "gtxl.dtd">
  <GTS id="simple">
    <Initial> // initial graph
      <Graph id="Private Server and public generator" edgeids="true" hypergraph="true" edgemode="undirected">
        <node id="n1"/> // nodes with unique ids
        <node id="n2"/>
        <rel id="ida"> // edge with unique ids
          <attr name="label"> // label of an edge is
            <string>A</string> // written as a string
          </attr>
          <attr kind="attr-int" name="a1"> // first attribute of type integer with value 10
            <string>10</string> // written as a string
          </attr>
          <attr kind="attr-int-str" name="a2"> // second attribute is a tuple integer-string
            <string>[5,'ab']</string> // with a value written as a string
          </attr>
          <relend id="spriv" target="n1" startorder="0" /> // connections of an edge
          <relend id="spriv" target="n2" startorder="1" />
        </rel>
      </Graph>
    </Initial>

    <Rule id="Rule1"> // first rule
      <precondition> // precondition = guard expression
        <condition>
          <attrCondition>
            <string>x>10 AND y>0</string> // expression itself as a string
          </attrCondition>
        </condition>
      </precondition>

      <LHS> // left-hand side of the rule
        <RuleGraph id="lr1">
          <Graph id="lgraph1" edgeids="true" hypergraph="true" edgemode="undirected">
            <node id="nl1"/> // nodes
            <node id="nl2"/>
            <rel id="la1"> // edge
              <attr name="label">
                <string>A</string> // label
              </attr>
              <attr kind="attr-int" name="a1"> // attributes are now from the term algebra
                <string>x</string> // here the value is x
              </attr>
              <attr kind="attr-int-str" name="a2"> // tuple
                <string>[y,z]</string>
              </attr>
              <relend id="lspriv1" target="nl1" startorder="0" />
              <relend id="lspriv2" target="nl2" startorder="1" />
            </rel>
          </Graph>
        </RuleGraph>
      </LHS>
    </Rule>
  </GTS>
```



```

    </Graph>
  </RuleGraph>
</LHS>

<RHS>                                // right-hand side of the rule
  <RuleGraph id="rr1">
    <Graph id="rgraph1" edgeids="true" hypergraph="true" edgemode="undirected">
      <node id="nr1"/>                // nodes
      <node id="nr2"/>
      <rel id="ra1">                  // first edge
        <attr name="label">
          <string>A</string> // label A
        </attr>
        <attr kind="attr-int" name="a1"> // attribute from the term algebra
          <string>x+5</string>          // value
        </attr>
        <attr kind="attr-int-str" name="a2">
          <string>[y-3,z.'a']</string>
        </attr>
        <relel id="rspriv1" target="nr1" startorder="0" /> // two connections to nodes
        <relel id="rspriv2" target="nr2" startorder="1" />
      </rel>

      <rel id="rb2">                // second edge
        <attr name="label">
          <string>B</string> // label B
        </attr>
        <attr kind="attr-int" name="a3"> // attribute from the term algebra
          <string>x+y</string>          // value
        </attr>
        <relel id="rspriv3" target="nr2" startorder="0" /> // only one connection
      </rel>
    </Graph>
  </RuleGraph>
</RHS>

  <Mapping id="cps_mapping">          // interface between left and right hand-sides of the rule
    <MapElem from="nl1" to="nr1" />   // as a map on nodes
    <MapElem from="nl2" to="nr2" />
  </Mapping>

</Rule> // end of the rule
</GTS>  // end of GTS

```

Appendix C

Example in new GTXL format

In this section we present a commented description of an AGTS from Section 4.2 (Fig. 4.3) in the new GTXL format. This format was proposed in [72] and is used for example in the AGG tool [104]. In August 2002 this format is used as an alternative to the GTXL format described in the last chapter. We also use the same GTS from Part 4.2 in order to illustrate its usage.

The main difference of the new GTXL format compared to the previous one is the description of a type graph which is here called "Schema Graph". Here, one describes the map from possible labels to the corresponding data types. For example, if an edge has the label "A" then it should have the attributes "a1" and "a2" of types integer and [integer,string] correspondingly. Later, in the concrete hypergraphs, only the values of attributes will be specified.

The next speciality of the new GTXL format is that each rule is given as a composition of three parts: "preserved", "deleted" and "created". In the part "preserved" one specifies a rule interface, which is in our case discrete and consist only of preserved nodes. In the "deleted" ("created") part one should indicate the nodes and edges from the left-hand side (right-hand side) of the rule which are not preserved in the interface.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gtxl SYSTEM "gtxl.dtd">
<gtxl xmlns:xlink="http://www.w3.org/1999/xlink" xmlns:xalan="http://xml.apache.org/xalan"
xmlns:xs1t="http://xml.apache.org/xslt">
  <graph edgeids="true" id="Schema Graph"> // Schema Graph is a type graph of GTS
    <node id="stdNode"> // There is only one type of nodes
      <graph edgeids="true">
        <rel id="A"> // Declaration of an edge with a label "A"
          <attr name="a1" kind="AttrType"> // Declaration of an attribute "a1"
            <string>int</string> // of type integer
          </attr>
          <attr name="a2" kind="AttrType"> // Declaration of an attribute "a2"
            <string>int-str</string> // of type integer-string
          </attr>
          <relel target="stdNode" startorder="0"/> // Connections of all edges labelled with "A"
          <relel target="stdNode" startorder="1"/> // are the same
        </rel>
        <rel id="B"> // Declaration of an edge with a label "B"
          <attr name="a1" kind="AttrType"> // Declaration of an attribute "a3"
            <string>int</string> // of type integer
          </attr>
          <relel target="stdNode" startorder="0"/> // Only one connection
        </rel>
      </graph>
    </node>
  </graph>
  <gts id="I2" approach="DP0"> // Start of GTS
    <type xlink:type="simple" xlink:href="#SchemaGraph"/> // Link to the schema graph
```

```

<initial>                                     // Initial graph

<graph id="ini" edgeids="true" hypergraph="true" edgemode="undirected">
  <node id="v1">                               // First node
    <type xlink:href="stdNode"/>               // linked to the Schema graph
  </node>
  <node id="v2">                               // Second node
    <type xlink:href="stdNode"/>
  </node>

  <rel id="r1">                                // Edge description
    <type xlink:href="A"/>                     // Type "A"
    <attr name="a1">                           // Attribute "a1"
      <string>10</string>                     // with value 10
    </attr>
    <attr name="a2">                           // Attribute "a2"
      <string>[5,'ab']</string>
    </attr>
    <relend id="relend1" target="v1"/>         // Connections
    <relend id="relend2" target="v2"/>
  </rel>

</graph>
</initial>

<rule name="create proc" id="Rule1">           // Description of rule
  <precondition>                               // precondition = guard expression
    <condition>
      <attrCondition>
        <string>x>10 AND y>0</string>         // expression itself as a string
      </attrCondition>
    </condition>
  </precondition>

  <preserved>                                  // Usually the discrete interface is preserved
  <graph id="G1">                              // i.e., only nodes
    <node id="N1">
      <type xlink:href="stdNode"/>
    </node>
    <node id="N2">
      <type xlink:href="stdNode"/>
    </node>
  </graph>
</preserved>

  <deleted>                                    // To delete all edges from
  <rel id="Rel_1">                             // the left-hand side of the rule
    <type xlink:href="A"/>                     // Edge "A"
    <attr name="a1">                           // Link to the Schema Graph
      <string>x</string>                       // Attribute "a1"
    </attr>                                   // with value x
    <attr name="a2">                           // Attribute "a2"
      <string>[y,z]</string>
    </attr>
    <relend id="Rel1_relend1" target="N1"/>     // Connections to N1
    <relend id="Rel1_relend2" target="N2"/>     // and N2
  </rel>
</deleted>

  <created>                                    // To create are all edges from
  <rel id="Rel2">                              // the right-hand side of the rule
    <type xlink:href="A"/>                     // Edge "A"
    <attr name="a1">                           // Attribute "a1"
      <string>x+5</string>                     // with value x+5
    </attr>
    <attr name="a2">                           // Attribute "a2"
      <string>[y-3,z,'a']</string>
    </attr>
    <relend id="Rel2_relend1" target="N1"/>
    <relend id="Rel2_relend2" target="N2"/>
  </rel>

  <rel id="Rel3">                             // Edge "B"
    <type xlink:href="B"/>

```

```
<attr name="a3">                                // Attribute "a3"
  <string>x+y</string>                          // with value x+y
</attr>
<releud id="Rel2_releud1" target="N2"/>
</releud>

</created>

</rule>

</gts>
</gtxl>
```

Appendix D

Example in GXL format

In this section we present a commented description of a Petri graph obtained as an over-approximation for the example from Section 4.2 (Fig. 4.3). The hypergraph component consists of two edges labelled "A" and "B" and two nodes. In the Petri net component there is only one transition corresponding to the rule "Rewriting Rule". Note that the guard of the transition is exactly the same as the guard of the rule.

```
<?xml version="1.0"?>
<gxl>
  <graph id="augur2.out" hypergraph="true" edgemode="undirected">
    <node id="_22"> // a node can be either a hypergraph node or a transition
      <attr name="vertex"/> // this is a hypergraph node
      <attr name="iota_mapping"> // iota match from the initial graph
        <string>n1</string>
      </attr>
    </node>
    <node id="_23"> // node
      <attr name="vertex"/> // hypergraph node
      <attr name="iota_mapping">
        <string>n2</string>
      </attr>
    </node>
    <rel id="_24"> // first edge
      <attr name="label">
        <string>A</string> // labelled with "A"
      </attr>
      <attr name="initial_marking"> // initial marking of the edge
        <token>
          <int>1</int> // one token
          <attr kind="attr-int" name="a1"> // attribute "a1" of the token
            <string>10</string> // value
          </attr>
          <attr kind="attr-int-str" name="a2"> // attribute "a1" of the token
            <string>[5,'ab']</string> // value
          </attr>
        </token>
      </attr>
      <attr name="iota_mapping"> // iota match for the edge
        <string>ida</string> // id in the initial graph
      </attr>
      <relend target="_22" role="vertex" startorder="0"/> // connections
      <relend target="_23" role="vertex" startorder="1"/>
      <relend target="_26" role="postset"> // postset for the transition with id="_26"
        <attr name="weight"> // weight on the arc is 1
          <int>1</int>
        </attr>
        <attr kind="attr-int" name="a1"> // attribute from term algebra on the arc
          <string>x+5</string> // value
        </attr>
        <attr kind="attr-int-str" name="a2"> // second attribute
          <string>[y-3,z.'a']</string>
        </attr>
      </relend>
      <relend target="_26" role="preset"> // postset
```

```

    <attr name="weight">
      <int>1</int>
    </attr>
    <attr kind="attr-int" name="a1">          // first attribute
      <string>x</string>
    </attr>
    <attr kind="attr-int-str" name="a2">      // second attribute
      <string>[y,z]</string>
    </attr>
  </relend>
</rel>
<rel id="_25">                               // second edge
  <attr name="label">
    <string>B</string>                       // labelled with "B"
  </attr>
  <attr name="initial_marking"/>             // not marked initially
  <attr name="iota_mapping">                 // iota is empty
    <string></string>
  </attr>
  <relend target="_23" role="vertex" startorder="0"/> // connection
  <relend target="_26" role="postset"        // postset for the transition with id = "_26"
    <attr name="weight">
      <int>1</int>                          // weight on arc is 1
    </attr>
    <attr kind="attr-int" name="a3">         // attribute on arc
      <string>x+y</string>
    </attr>
  </relend>                                // not in the preset of any transition
</rel>
<node id="_26">                             // transition with id = "_26"
  <attr name="transition"/>
  <attr name="rule">
    <string>Rewriting Rule</string>         // name of the rule
  </attr>
  <attr kind="attr-bool">                   // guard function of the type boolean
    <string>x>10 AND y>0</string>          // value
  </attr>
  <attr name="mu_mapping">                 // mu match from a rule to the Petri graph
    <string>n1,22;n12,23;cps_lh_gpub,24;nr1,22;nr2,23;cps_rh_Sprv,24;idbr,25;</string>
  </attr>
  // match is based on ids
</node>
</graph>
</gxl>

```

Appendix E

Example of an SPL program

In this section we present an example of a program in SPL format, which can be used as an input for AUGUR 2 [105]. For this purpose the corresponding reader (SPL reader) should be switched on in the database of AUGUR 2 (see Section 6.3 and Section 6.4). The program will then be translated into an AGTS, which can be further approximated and analyzed. The details of the SPL language and its translation into AGTSs can be found in [105].

The program consists of the type declaration part, where a record consisting of one integer (`info_rec`) and list of such records (`list_rec`) are defined. Then the variables (`list`, `help` of type `pointer to list_rec` and `i` of type `integer`) are declared and initialized. After this the functionality of the program is defined: First, an infinite list of records is produced and then during the iteration through this list each element in the record will be increased by one.

For each pointer (`list` and `help`) an error rule will be added which should check if at some point of the program an assignment to the null-pointer is made. With the translation to AGTSs the special edge "error" will be created. The property to verify is that no error-edge will ever be created.

```
TYPE info_rec;  
TYPE list_rec IS RECORD (next: POINTER TO RECORD list_rec, info: POINTER TO  
RECORD info_rec);  
TYPE info_rec IS RECORD(value: INTEGER);  
  
VAR list: POINTER TO RECORD list_rec;  
VAR help: POINTER TO RECORD list_rec;  
VAR i: INTEGER;  
  
i:=0;  
NEW(list);  
NEW(list.info);  
help:=list;  
help.info.value:=i;  
  
WHILE(TRUE) DO  
  i:=i+1;  
  NEW(help.next);  
  help:=help.next;  
  NEW(help.info);  
  help.info.value:=i;  
OD;  
help:=list;  
WHILE(TRUE) DO  
  help.info.value:=help.info.value+1;  
  help:=help.next;  
OD;  
SKIP;
```

Although the translator from the SPL language into GTs (AGTs) has been developed and tested on some number of examples, still verification of further case studies remains a matter of future work.

Appendix F

Verification Example

In this section we present the verification protocol for the attributed example from Section 4, Fig. 4.4 (file `augur/example/attributed/conn2.xml` in the tool). We use here the command line version of the tool. The property we want to verify is that no edge labelled "Error" is created. We use modulo abstraction of attributes with the initial value $mod = 1$. This should be set in the database in the following way:

```
<Globals>
...
<mod_base val="1"/>
...
</Globals>

...

<algorithm name="storage" reusable="true">
  <label name="attribute_engine">
    <default algorithm="expression_engine_mod"/>
    <info>Here an attribute engine (abstraction) is set</info>
    <expert status="false"/>
  </label>
  ...
</algorithm>

...
```

First construct the approximated unfolding using the scenario `aunfold`:

```
> bin/augur -db=db/default.xml -sc=aunfold
example/attributes/conn2.xml work/PetriNet_0.xml

loading database
Running scenario ...
step 1:
searching matches.unfolding rule 9 (Cross Forward) ...done.
step 2:
searching matches.folding rule 9 (Cross Forward) ...done.
step 3:
searching matches..unfolding rule 10 (Cross Backward) ...done.
```

```

step 4:
searching matches.folding rule 9 (Cross Forward) ...done.
step 5:
searching matches.folding rule 9 (Cross Forward) ...done.
step 6:
searching matches.folding rule 9 (Cross Forward) ...done.
step 7:
searching matches...unfolding rule 11 (Create C) ...done.
step 8:
searching matches.folding rule 9 (Cross Forward) ...done.
step 9:
searching matches.folding rule 9 (Cross Forward) ...done.
step 10:
searching matches....unfolding rule 12 (Error) ...done.
step 11:
searching matches....calculation finished.
time used: 0.01 sec.
edges: 4, vertices: 1, transitions: 4

```

The computed Petri graph (Petri net and graph component) can be visualized using scenarios `pn2ps` and `hg2ps`. The initial graph and the rules can be visualized using the scenario `rules2ps`.

For example the visualization of the hypergraph can be done in the following way:

```

> bin/augur -sc=hg2ps -db=db/default.xml
work/HyperGraph_0.xml work/out.hg.ps

```

The result is saved in the postscript file "out.hg.ps".

Now we convert the regular expression `0'Error'0` describing the error edge having arity 0 and corresponding to the property to be verified in order to obtain a marking of a Petri graph. We use scenario `property2marking`.

```

> bin/augur -db=db/default.xml -sc=property2marking work/PetriNet_0.xml
work/regexpr work/marking

```

```

loading database
Running scenario ...
info: Hypergraph: vertices: 101   edges: 102 <S>(101 ) 103 <P>(101 )
104 <Error>() 105 <C>(101 101 )
info: result: MinCover {
Path 13926736 -> 164505152 f: 104 |-> 1
}

```

```

FORMULA ( Error_104 >= 1 )

```

Here the regular expression is saved in the file "work/regexpr" and the resulting marking is written to "work/marking". The regular expression can also be more complex, for example `'A'C*'B'` representing the property that no chain of "C"s will be created from an edge "A" to "B".

We continue with our example. In this case the file `marking` describing markings that should *not* be coverable looks as follows:

```

MARKING Error_104: 1;

```

If this marking is not coverable, then the verified property is true. Now we are ready to check if this marking is coverable in the underlying Petri net. To this aim we call the scenario `cover` with a standard coverability algorithm, which checks whether a marking can be covered, in the following way:

```
> bin/augur -db=db/default.xml -sc=cover
work/PetriNet_0.xml work/marking work/transpath
```

```
loading database
Running scenario ...
Searching the trace to id=5
write counter-example to augur/work/transpath
```

```
The Final Marking(s) are coverable
time used: 0 sec.
```

We can see that the marking is coverable in the Petri net and the trace is saved in the file "work/transpath":

```
Error
109
% x(int): 0
```

But this does not mean that the verified property is false, since we work with an over-approximation. Let us now call counterexample-based abstraction refinement with scenario `refinement` in order to obtain a more precise approximation.

```
> bin/augur -db=db/default.xml -sc=refinement example/attributes/conn2.xml
work/PetriNet_0.xml work/transpath work/sample.xml work/PetriNet_1.xml
```

```
loading database
Running scenario ...
The example is spurious
Transition number 1, id: 109, name: Error,
could not be fired
Structural refinement is needed
equivalence class: nodes ( 2 3 4 )
optimized sample size: 1
step 1:
searching matches.unfolding rule 9 (Cross Forward) ...done.
...
step 33:
searching matches....unfolding rule 12 (Error) ...done.
step 34:
searching matches....calculation finished.
time used: 0.25 seconds
edges: 9, vertices: 3, transitions: 13
```

Now the refined over-approximation is saved in the file "work/PetriNet_1.xml". From the output of the tool it can be seen that the refinement step was structural. If we repeat all earlier verification steps with scenarios `property2marking` and `cover`, then we will see, that the undesirable situation is still possible in the Petri net obtained from the first refinement step.

In this case scenario `cover` gives the following answer

```

loading database
Running scenario ...
Searching the trace to id=11
write counter-example to augur/work/transpath

```

```

The Final Marking(s) are coverable
time used: 0.01 sec.

```

This means that the marking is still coverable and the new counterexample is

```

Cross Backward
618
% x(int): 0, y(int): 0

```

```

Error
629
% x(int): 0

```

If we start the counterexample-based abstraction refinement again then the new output is

```

> bin/augur -db=db/default.xml -sc=refinement example/attributes/conn2.xml
work/PetriNet_1.xml work/transpath work/sample.xml work/PetriNet_2.xml

```

```

loading database
Running scenario ...
The example is spurious
Transition number 1, id: 618, name: Cross Backward,
could not be fired
Refinement of attributes is needed
(1) Attributes have been refined
Spurious counter-example is eliminated

```

We see that in this case the refinement of attribute abstraction was needed and the attributes have been successfully refined. After the refinement step the modulo abstraction value becomes 2 and the repetition of verification steps with scenarios `property2marking` and `cover` shows us that the marking corresponding to the edge "Error" is no more coverable:

```

loading database
Running scenario ...

The Final Marking(s) are not coverable.
time used: 0.46 sec.

```

This means we have successfully verified the example. All verification steps above can be reproduced using the graphical user interface of AUGUR 2 (see Section 6.4).